UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Implementación de algoritmos IA en GPUs

Roberto Moreno Valderrama
Carlos Aguirre Maeso

JUNIO 2017

IMPLEMENTACIÓN DE ALGORITMOS IA EN GPUS

AUTOR: Roberto Moreno Valderrama
TUTOR: Carlos Aguirre Maeso

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2017

Resumen (castellano)

Este Trabajo Fin de Grado trata de explorar uno de los aspectos que menos avance tecnológico ha sufrido dentro de los videojuegos en comparación a otros elementos del mismo: la inteligencia artificial.

Siendo una de las tareas que más cómputo en el procesador consume, se ha tratado de minimizar su tiempo de proceso haciéndola más simple y usando menos elementos que la requieran. En este trabajo buscamos ayudarnos de la GPU, componente principal en la ejecución de videojuegos, para repartir la carga del cálculo de la inteligencia artificial entre ella y el procesador de manera que, en un futuro, pueda ser posible la creación de IAs más complejas o manejo de un mayor número de instancias de ellas dando lugar a videojuegos más interesantes.

Uno de los algoritmos más utilizados en este ámbito es la búsqueda A*. Éste es usado como un recurso para *pathfinding*. Es usual que, en un videojuego, necesitemos que el ordenador mueva varias unidades con una determinada IA a través de un mapa. Además, la búsqueda A* es un algoritmo no recursivo y ampliamente estudiado, por lo que es un candidato ideal para nuestro desarrollo.

Para implementar algoritmos en GPU haremos uso de la especificación OpenCL que distintas manufactureras de GPUs se han comprometido a implementar en sus dispositivos. Está especificación nos permite comunicarnos con la GPU de manera sencilla y general, sin la necesidad de usar elementos gráficos. Buscamos de qué maneras es posible hacer uso la gran cantidad de hilos de la GPU con el algoritmo A*.

Después de la codificación de varias implementaciones, observamos que una implementación que hace uso de la GPU en las partes críticas del algoritmo (más concretamente en la búsqueda A*, en la expansión de nodos) produce una enorme aceleración del tiempo de ejecución de la búsqueda.

Esperamos que, en un futuro cercano, gracias a este tipo de estudios sobre algoritmos ampliamente usados en IAs de videojuegos y la creciente accesibilidad a la programación en GPU, pueda usarse este potente dispositivo para este tipo de propósitos y obtener videojuegos con mayor profundidad y complejidad.

Palabras clave (castellano)

búsqueda A, paralelización, inteligencia artificial, videojuegos, OpenCL*

Abstract (English)

This Bachelor Thesis explores one of the less technologically advanced videogame elements: artificial intelligence.

It often is a task which draws more computational power from CPU than other elements. For this reason, developers have tried to minimize its execution time making it less complex and using less elements ingame which use it. In this Project, we try to use the GPU, which is already the main computer component used by a videogame, to move compute load from CPU to GPU. We want to open a way to create complex AIs or manage a greater number of AI instances with this change.

One of the most used algorithms in the AI field is the A* search algorithm. It is a pathfinding tool. In a videogame, usually the computer moves several units across the map and it is there where A* is used. Moreover, A* search is a non recursive algorithm and it has been widely studied, therefore is a fantastic candidate for our Project.

We will work with the OpenCL specification to implement algorithms in GPU. The biggest GPU manufacturing companies have agreed to follow and implement in their devices this specification. We study which ways we can take advantage of the large number of threads a GPU spawns for our A* search algorithm.

After several implementations have been coded, we notice that a hybrid implementation where critical spots of the algorithm are parallelized (specifically, node expansion) in GPU outputs a great speedup of the search A* execution time.

We hope that thanks to this kind of studies about algorithms widely used in videogames and accessibility to program in GPU there will be videogames deeper and more complex in the near future.

Keywords (inglés)

A search, parallelization, artificial intelligence, videogames, OpenCL*

Agradecimientos

A mi familia y amigos.

INDICE DE CONTENIDOS

1 Introducción 1
1.1 Motivación 1
1.2 Objetivos y enfoque 2
2 Estado del arte 3
3 Diseño 5
3.1 Búsqueda A* 5
3.2 Herramientas 7
3.2.1 Lenguaje de programación: C++ y OpenCL 7
3.2.2 Framework: Visual Studio 2015 7
3.2.3 Hardware 7
3.2.4 Representación de datos: RStudio 7
3.3 Metodología de la implementación GPU de algoritmos 8
3.3.1 Búsqueda A* 9
4 Resultados 10
4.1 Metodología para las pruebas 10
4.2 Búsqueda A* 13
4.2.1 Expansión de nodos en GPU: primera implementación (A1) 13
4.2.2 Expansión de nodos en GPU: segunda implementación (A2) 15
4.2.3 Expansión de nodos en GPU: tercera implementación (A3) 16
4.2.4 A* íntegro en GPU: expansión de nodos no paralelizada (B1) 19
4.2.5 A* íntegro en GPU: expansión de nodos paralelizada (B2) 20
4.2.6 Múltiples instancias en CPU y GPU (C) 24
5 Conclusiones y trabajo futuro 26
5.1 Conclusiones 26
5.1.1 Aspectos negativos 26
5.1.2 Aspectos positivos 27
5.2 Trabajo futuro 27
Referencias 29
Glosario 29
Anexos I
A Implementaciones en detalle I
Estructuras I
Funciones adicionales II
Kernel A1 II
Kernel A2 III
Kernel A3 IV
Kernel B1 VI
Kernel B2 VIII
Kernel C XII

INDICE DE FIGURAS

FIGURA 3-1 PSEUDOCÓDIGO DEL ALGORITMO DE BÚSQUEDA A* 5

FIGURA 3-2 DISTINTAS MANERAS DE PARALELIZAR EL ALGORITMO A* 9

FIGURA 4-1 LONGITUD MEDIA DE LOS CAMINOS ENCONTRADOS CON A* EN GRADOS DE 50 NODOS 11

FIGURA 4-2 LONGITUD MEDIA DE LOS CAMINOS ENCONTRADOS CON A* EN GRADOS DE 500 NODOS 11

FIGURA 4-3 PRUEBA DE LA IMPLEMENTACIÓN A1 CON 50 NODOS 13

FIGURA 4-4 PRUEBA DE LA IMPLEMENTACIÓN A1 CON 100 NODOS 14

FIGURA 4-5 PRUEBA DE LA IMPLEMENTACIÓN A1 CON 200 NODOS 14

FIGURA 4-6 PRUEBA DE LA IMPLEMENTACIÓN A2 CON 50 NODOS 15

FIGURA 4-7 PRUEBA DE LA IMPLEMENTACIÓN A2 CON 100 NODOS 15

FIGURA 4-8 PRUEBA DE LA IMPLEMENTACIÓN A2 CON 500 NODOS 16

FIGURA 4-9 PRUEBA DE LA IMPLEMENTACIÓN A3 CON 200 NODOS 17

FIGURA 4-10 PRUEBA DE LA IMPLEMENTACIÓN A3 CON 500 NODOS 17

FIGURA 4-11 PRUEBA DE LA IMPLEMENTACIÓN A3 CON 1000 NODOS 18

FIGURA 4-12 IMPLEMENTACIONES DEL GRUPO A 18

FIGURA 4-13 PRUEBA DE LA IMPLEMENTACIÓN B1 CON 100 NODOS 19

FIGURA 4-14 PRUEBA DE LA IMPLEMENTACIÓN B1 CON 200 NODOS 20

FIGURA 4-15 COMPARACIÓN DE B2 CON B1 SOBRE UN GRAFO DE 50 NODOS. 21

FIGURA 4-16 COMPARACIÓN DE B2 CON B1 SOBRE UN GRAFO DE 100 NODOS. 21

FIGURA 4-17 COMPARACIÓN DE B2 CON B1 SOBRE UN GRAFO DE 200 NODOS. 22

FIGURA 4-18 COMPARACIÓN DE B2 CON A3 SOBRE UN GRAFO DE 100 NODOS. 22

FIGURA 4-19 COMPARACIÓN DE B2 CON A3 SOBRE UN GRAFO DE 200 NODOS. 23

FIGURA 4-20 COMPARACIÓN DE B2 CON A3 SOBRE UN GRAFO DE 500 NODOS. 23

FIGURA 4-21 EJECUCIÓN DE 500 INSTANCIAS DE LAS IMPLEMENTACIONES CPU Y B1 EN UN GRAFO DE 50 NODOS. 24

FIGURA 4-22 EJECUCIÓN DE 500 INSTANCIAS DE LAS IMPLEMENTACIONES CPU Y B1 EN UN GRAFO DE 100 NODOS. 25

FIGURA 4-23 EJECUCIÓN DE 500 INSTANCIAS DE LAS IMPLEMENTACIONES CPU Y B1 EN UN GRAFO DE 200 NODOS. 25

FIGURA 4-24 COMPARACIÓN DE CPU Y B1 EN UN GRAFO DE 50 NODOS Y SPARSEFACTOR = 0.4... 26

INDICE DE TABLAS

TABLA 1 ESPECIFICACIONES HARDWARE 7

1 Introducción

Lo que empezó como un componente opcional y que sólo unos pocos adquirirían para sus ordenadores alrededor de los años 70s y 80s, se ha convertido en otra de las especificaciones fundamentales que todo distribuidor y manufacturador debe preocuparse de añadir. Las *Graphics Processing Unit* o GPU son unidades con una gran cantidad de procesadores segmentados paralelamente para que puedan trabajar durante una misma unidad de tiempo [5]. A diferencia de las *Central Processing Unit* o CPU que poseen un número considerablemente menor de unidades de procesamiento, la información que se puede transmitir entre los distintos procesadores está más limitada debido a su arquitectura. Esta razón propició que las GPU se usasen para propósitos muy específicos. Como su propio nombre indica, se usaron para la representación de imágenes y gráficos por pantalla.

Debido a este particular uso, el desarrollo y mejora de este componente estaría (y sigue estando) fuertemente ligado al crecimiento de la industria centrada en crear productos gráficos: la industria de los videojuegos. A pesar de un inicio lento, los videojuegos fueron calando en la sociedad expandiéndose con famosas videoconsolas de entretenimiento como la Atari 2600 o la ColecoVision o con sus sucesoras Nintendo Entertainment System (NES) o la Sega Master System. Desde ese momento, las desarrolladoras de videojuegos trataron de ofrecer a sus consumidores productos cada vez más elaborados y complejos que requerían componentes más potentes. Más tarde, nacerían empresas como NVidia Corporation y ATI Technologies que se centraron exclusivamente en satisfacer dicha demanda.

Un tiempo más tarde, los *Personal Computer* o PC se convirtieron en una realidad y desde entonces la cantidad de personas con acceso a uno ha ido en aumento [7]. Estos factores han hecho que la presencia de GPUs se casi ubicua y sea considerado otro componente fundamental más en ordenadores.

Con el enorme desarrollo que sufrieron las GPUs, se comenzó a investigar si podían tener otro propósito. Se llegó a la conclusión de que existen multitud de campos donde se puede dar uso a la potencia bruta en paralelo que ofrecen. Por ello, se creó el concepto de *General Purpose computing for GPU* o GPGPU para aquellos propósitos ajenos a la computación de gráficos [10]. Junto con él se crearon diversas herramientas que facilitasen a programadores el paso de un paradigma de programación secuencial a uno paralelizable.

1.1 Motivación

Los videojuegos se han convertido hoy en día en una de las grandes industrias del entretenimiento obteniendo más ingresos que industrias tradicionales y más asentadas como la filmografía y la música [6]. Los productos creados por esta industria se refinan y pulen hasta explotar al máximo los recursos disponibles creando año tras año videojuegos con tecnología puntera.

Debido a que los videojuegos, en su forma más básica, son un conjunto de vértices que hay que calcular constantemente su posición para poder mostrarlos al usuario, los esfuerzos de optimización de recursos se han centrado en esos cálculos. Pero desde que se creó uno de los primeros videojuegos, *Pong*, la Inteligencia Artificial ha estado presente en una gran parte de ellos. Y es el elemento que más computo necesita que aún no es propiamente

paralelizado. Esto puede deberse a varias razones: la codificación de IAs en GPU no está extendida, existen multitud de algoritmos para IA ya implementados en CPU y han sido investigados y probados durante bastante tiempo. No todos los videojuegos se benefician de una buena IA o necesitan una, pero aquellos que sí, como por ejemplo los videojuegos de estrategia que deben gestionar cientos de pequeñas inteligencias constantemente, a veces se encuentran en la tesitura de eliminar lógica, y por tanto crear unidades menos inteligentes, con el fin de que la CPU no afecte al rendimiento total del producto.

Sabiendo que podemos darle otros usos a la GPU, nos parece interesante estudiar una implementación de algoritmos de IA usados frecuentemente en este tipo de videojuegos. Más concretamente, el algoritmo de búsqueda A* sigue siendo ampliamente usado como método de *pathfinding*.

1.2 Objetivos y enfoque

Los objetivos del proyecto se pueden detallar como los siguientes:

- Estudio y entendimiento de la programación en GPU con la herramienta OpenCL.
- Estudio e investigación del algoritmo de búsqueda A* para encontrar aquellas partes paralelizables e implementarlas en GPU.
- Comparar implementaciones del algoritmo en CPU y GPU y observar su rendimiento con distintos parámetros.

Existen multitud de algoritmos pensados directamente para ser ejecutados en GPU paralelamente y ya son altamente eficientes. No obstante, el punto interesante es intentar conseguir mejorar algoritmos ya codificados en multitud de juegos y conocidos ampliamente por programadores para, si se diese el caso, poder realizar una transición fácil de CPU a GPU.

2 Estado del arte

En los últimos años, grandes empresas del sector tecnológico, como Nvidia y Google, han creado grandes IAs usando aprendizaje automático mediante redes neuronales. Estas redes neuronales, por su propia naturaleza, son ideales para ser ejecutadas en grandes *grids* de GPUs. Aunque esto son algoritmos demasiado complejos y su único fin es el desarrollo de una IA centrada en temas muy concretos [8][9].

Existen algunos trabajos que ya exploran la posibilidad de usar la GPU con algoritmos como el A*:

Rafia Inam, de la Chalmers University of Technology [1], desarrolló su tesis de máster sobre este mismo tema en 2009. En ella realiza una implementación en CUDA (lenguaje de programación en GPU propietario de Nvidia) de A* para explorar los nodos del grafo. Sin embargo, se encuentra con limitaciones de hardware (en los últimos años la potencia en bruto de las GPU ha crecido enormemente) y se centra en grafos adaptados a un modelo "*rejilla"* (cada nodo tiene acceso a otros 4 nodos) propios de algunos videojuegos. Esto reduce su capacidad para paralelizar trabajo.

Más recientemente, Yichao Zhou y Jianyang Zeng de la Tsinghua University [2] realizaron una implementación con grafos generales. Obtuvieron resultados interesantes: el algoritmo paralelizado conseguía realizar más rápidamente aquellas tareas donde era necesario A*.

3 Diseño

3.1 Búsqueda A*

Uno de los algoritmos más conocidos y utilizados en el mundo de los videojuegos. Usado como método de *pathfinding*, búsqueda del camino más corto, ofrece la solución óptima con unos costes aceptables [3]:

- Worst time: O(|E|)
- Worst space: O(|V|)

donde E es el conjunto de aristas del grafo donde se produce la búsqueda y V el conjunto de vértices.

El algoritmo se basa en recorrer todos los caminos posibles hasta encontrar el más corto. En la práctica llevaría demasiado tiempo por tanto es ayudada por una *heurística* para elegir aquellos nodos que parecen más prometedores de llegar al nodo objetivo más rápidamente. Esta *heurística* o estimación no es estática, según el grafo con el que se esté trabajando es necesario el uso de una que ayudé al algoritmo A* a ser lo más eficiente posible.

La implementación usada es la dada por el siguiente pseudocódigo [4]:

```
      // A*
1:    initialize the open list
2:    initialize the closed list
3:    put the starting node on the open list (you can leave its f at zero)
-
4:    while the open list is not empty
5:        find the node with the least f on the open list, call it "q"
6:        pop q off the open list
7:        generate q's 8 successors and set their parents to q
8:        for each successor
9:            if successor is the goal, stop the search
10:           successor.g = q.g + distance between successor and q
11:           successor.h = distance from goal to successor
12:           successor.f = successor.g + successor.h
-
13:           if a node with the same position as successor is in the OPEN list \
-                 which has a lower f than successor, skip this successor
14:           if a node with the same position as successor is in the CLOSED list \
-                 which has a lower f than successor, skip this successor
15:           otherwise, add the node to the open list
16:       end
17:       push q on the closed list
18:   end
```

Figura 3-1 Pseudocódigo del algoritmo de búsqueda A*

3.2 Herramientas

3.2.1 Lenguaje de programación: C++ y OpenCL

Para poder programar en GPU son necesarias ciertas librerías que se encarguen de la comunicación entre el *host* (CPU) y nuestro *device* (GPU). Al principio programar para GPU era más complicado pues no existían, sin embargo, con la aparición de GPGPU, se empezaron a crear y de tal manera surgieron tales como CUDA para las tarjetas gráficas de Nvidia y OpenCL que es una especificación creada por *Khronos Group* para que cada empresa manufacturera de tarjetas gráficas implemente una serie de librerías que la cumplan para sus propios modelos [11] [12].

En nuestro caso, las especificaciones de nuestro ordenador únicamente nos permiten acceder a las librerías de OpenCL creadas por AMD y distribuidas en su *AMD APP SDK*. Pertenecen a C++, lenguaje desde donde llamaremos a las funciones necesarias para la ejecución de un *kernel*. Un *kernel* posee muchas similitudes con la función *main* de un programa de C o C++. El *kernel* será la primera función que se llame dentro de la GPU.

3.2.2 Framework: Visual Studio 2015

Aunque la programación sean C++ y se pueda compilar con la utilidad *gcc* de Linux, es recomendable realizar programación en tarjeta gráfica en Windows. Históricamente se ha preferido este sistema operativo para tareas de este tipo.

Luego, la elección de Visual Studio para desarrollar y ejecutar el código se debe a que AMD en su AMD APP SDK proporciona una serie de ejemplos y proyectos funcionales de OpenCL para ayudar a los iniciados a entender cómo usar sus librerías. Además, la configuración de Visual Studio de los ejemplos no presenta ningún error y hace que el inicio a la programación de OpenCL sea rápida y cómoda.

3.2.3 Hardware

El hardware del ordenador donde se ha realizado toda la codificación y pruebas del proyecto es el siguiente:

Tipo dispositivo	Nombre	Frecuencia (GHz)	GFLOPS
CPU	AMD Phenom II x4 970	3.6	~45 [13]
GPU	AMD R9 Fury	1.05	~7200 [14]

Tabla 1 Especificaciones hardware

3.2.4 Representación de datos: RStudio

Para el manejo de datos y trazado de gráficos existen multitud de librerías en distintos lenguajes. Hemos decidido usar R junto con el *framework* RStudio debido a dos razones: una gran flexibilidad a la hora de interpretar los datos y, previamente al proyecto, ya estaba familiarizado con el entorno. Aparte, posee distintas librerías como *ggplot2* que facilita aún más la generación de gráficas y representación de datos.

3.3 Metodología de la implementación GPU de algoritmos

La metodología para realizar la implementación está compuesta por dos partes. La primera parte es la *codificación* y ha seguido los siguientes pasos:

- Partiendo del código original de CPU, encontrar el bucle más interno del algoritmo.
- Estudiar si las operaciones del bucle se pueden realizar paralelamente.
- Si se cumple el anterior punto, crear un *kernel* para que el código del bucle se ejecute en GPU en vez de CPU.

En este punto, la implementación es funcional y arroja los mismos resultados que su homólogo en CPU.
Por desgracia, la implementación en GPU de algoritmos no está libre de un costo computacional adicional ya que hay que establecer una comunicación entre la CPU y la GPU. Ya que no sería justo comparar ambas cuando una está sufriendo de un retraso temporal adicional, es necesario minimizar dicho costo (también denominado *overhead*) llamando el mínimo de funciones necesarias y sólo en partes concretas del código para evitar esparcir y mezclar el código referente a la GPU con el original. Esta es la segunda parte de la implementación, la *optimización*.

En OpenCL los pasos necesarios para ejecutar un *kernel* son los siguientes:

- Buscar el *device* donde se ejecutará el código (la GPU en este caso).
- Crear un contexto, crear una pila de comandos (*commandQueue*) y un programa dado un archivo .cl.
- Crear buffers para aquellos argumentos del *kernel* que sean arrays.
- Establecer los argumentos del *kernel*.
- Crear *kernel*.

Gracias a como está pensado la introducción de los parámetros al *kernel* antes de su ejecución, es posible crear el *kernel* y más tarde establecer los valores de los argumentos. De esta manera, surgen dos zonas en el algoritmo para insertar código OpenCL: la primera es al principio del algoritmo (o antes de su ejecución). Todos los elementos que no se vean alterados a lo largo del algoritmo deberán ser inicializados aquí. Eso es: búsqueda *device*, contexto, pila de comandos, programa, creación *kernel*, creación de los búferes (es menos costoso vaciar y llenar un búfer que estar constantemente creándolos), establecer argumentos inmutables y establecer argumentos de búfer. La segunda zona, antes de la ejecución del kernel, se deberá emplear sólo y estrictamente para insertar nuevos valores en los argumentos del *kernel*, búfer o variable. Después de la ejecución, se recogerán los resultados y liberarán los recursos reservados para el kernel.

Siguiendo estas directrices es posible implementar partes de algoritmos en paralelo de manera relativamente sencilla. Aunque si se trata de un algoritmo más secuencial donde los datos están en constante dependencia de sus estados anteriores la implementación es mucho más costosa pues entrará en juego la sincronización de información dentro del propio *kernel*. O, como en muchos otros casos, es imposible de implementar o no merece la pena ya que perjudica al rendimiento.

3.3.1 Búsqueda A*

La búsqueda A* es un buen candidato para obtener resultados satisfactorios de una implementación en paralelo. Es un algoritmo no recursivo donde las iteraciones de su bucle más interno, encargado de la expansión de nodos, son independientes las unas de las otras.

Las implementaciones desarrolladas se pueden dividir en tres grupos:

- **Grupo A:** Es el objetivo original. Tratar de paralelizar ciertas partes del algoritmo. Dentro de él surgen tres implementaciones, cada una implementado un poco más de la anterior:

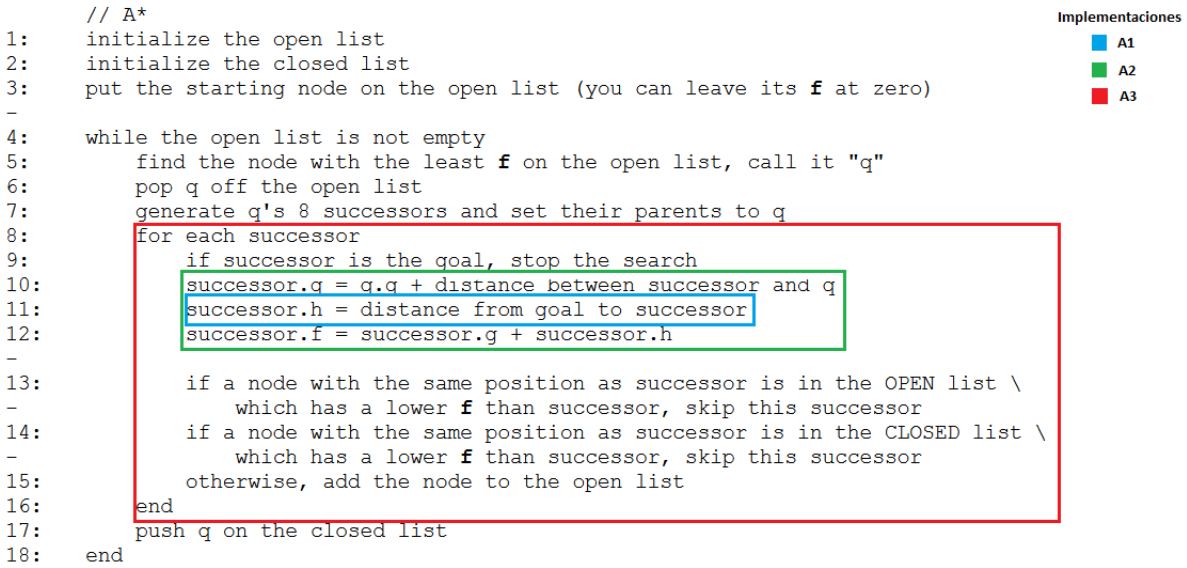

Figura 3-2 Distintas maneras de paralelizar el algoritmo A*

A la GPU se le pasará la información necesaria relacionado con la búsqueda en el momento actual. El *kernel* será inicializado con un valor igual al número de nodos sucesores que se quieren expandir. Gracias al funcionamiento de las GPUs, pueden ser llamados un número ilimitado de hilos para ejecutar un *kernel*. La GPU se encargará de llamar en paralelo al máximo número posible y ejecutará los demás en cuanto pueda. Por estos motivos no existen limitaciones adicionales al trabajar en GPU en estos casos.

- **Grupo B:** Tras haber conseguido el objetivo inicial, pensamos en otras maneras de servirnos de la GPU al buscar con A* y surgieron dos implementaciones centradas en ejecutar el algoritmo íntegro en el *kernel*.

  La primera (**B1**) usará un único hilo de la GPU, por lo que será una implementación análoga a la estándar en CPU. Mientras tanto, en la segunda (**B2**), paralelizamos la anterior implementación. Es decir, uno de los hilos de la GPU será el *padre* o principal que se encargará de dictar las fases del algoritmo y sincronizar las variables globales de la búsqueda. Por otro lado, los demás hilos se encargarán de expandir los nodos que les correspondan cuando el hilo principal lo indique.

- **Implementación C:** Por último, pensamos que es interesante explorar las posibilidades de ejecutar una gran cantidad de instancias del algoritmo

paralelamente en GPU. Para esta implementación nos basaremos en lo desarrollado en el anterior grupo (B1). Se usarán varios hilos en vez de sólo uno.

4 Resultados

En esta sección mostraremos las pruebas realizadas y explicaremos los resultados obtenidos.

4.1 Metodología para las pruebas

En cada una de las pruebas compararemos los tiempos de ejecución de distintas implementaciones. Para conseguir los resultados más fiables, siempre que una de ellas realizaba una búsqueda desde un nodo *start* a un nodo *end* en un determinado grafo, los otros algoritmos con cuáles está siendo comparado lo ejecutarán también. De esta manera, tenemos los siguientes parámetros de entrada para nuestros test:

- ***nnodos***: La cantidad de nodos que tendrá el grafo a crear. Los valores usados han sido:

  { 50, 100, 200, 500 }

  Y en ciertas ocasiones 1000 cuando 500 no sea suficiente para ver la diferencia o simplemente para acentuar la diferencia entre implementaciones.

- ***sparsefactor[]***: un array de valores *float* que indicarán como de denso e interconectado queremos que esté nuestro grafo. Los valores usados han sido los siguientes:

  [0.001, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6]

  A partir de 0.6 es muy probable que el camino que se encuentre sea directo, por lo que es menos interesante de estudiar. A continuación, mostramos una gráfica representando las distintas longitudes medias de los caminos resultantes con *sparsefactor* variable:

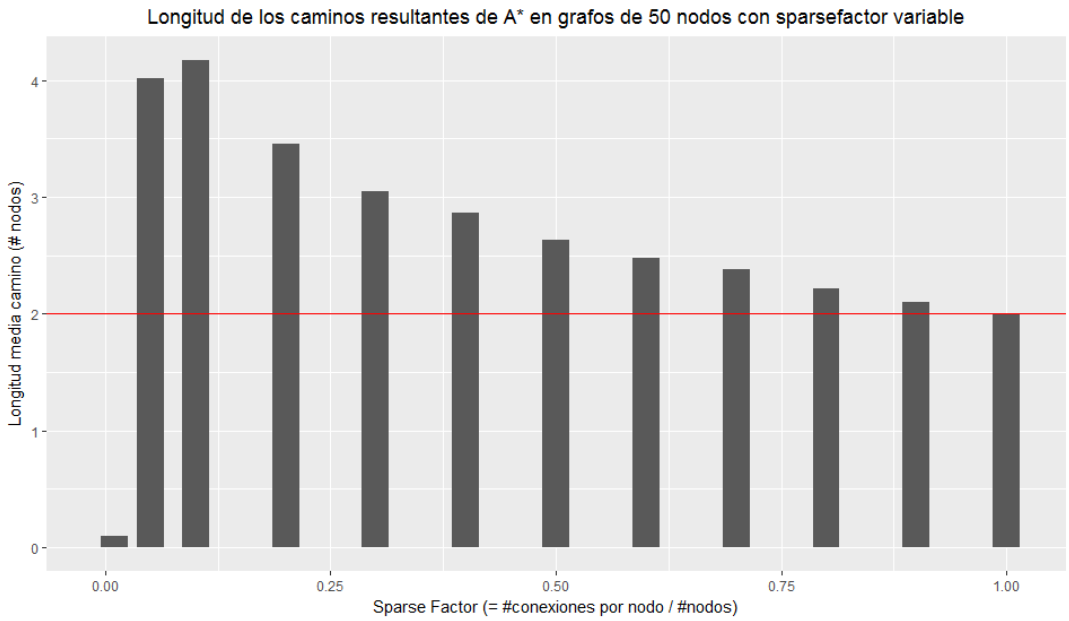

Figura 4-1 Longitud media de los caminos encontrados con A* en grados de 50 nodos

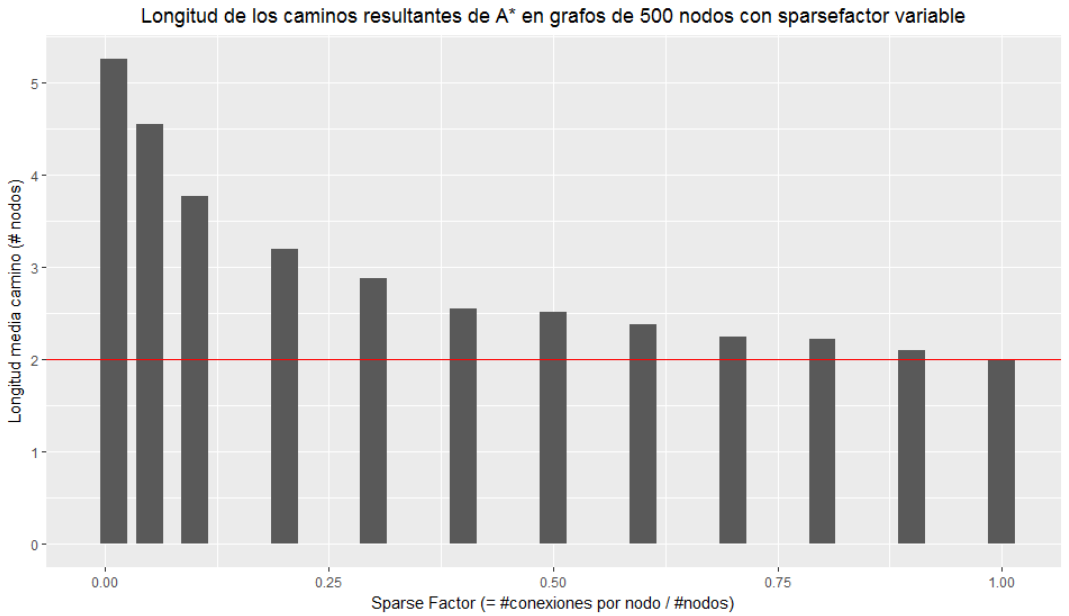

Figura 4-2 Longitud media de los caminos encontrados con A* en grados de 500 nodos

A partir de 0.6 ya son caminos directos (2 nodos) en su mayoría y, por tanto, nos darán información menos interesante.

- ***reps***: en un mismo grafo, dependiendo de los nodos elegidos como inicial y final, puede variar enormemente el tiempo de ejecución. Por ello, para evitar anomalías en los valores, repetiremos la ejecución de la búsqueda de un grafo con cada valor de *nnodos* y *sparsefactor* una cantidad *reps* de veces y con valores de los nodos

inicial y final distinto. El valor usado ha sido 50, suficiente para suavizar las gráficas sin consumir un tiempo excesivo.

*4.2 Búsqueda A**

4.2.1 Expansión de nodos en GPU: primera implementación (A1)

La GPU se encarga únicamente del cálculo de la heurística de cada nodo sucesor en esta implementación.

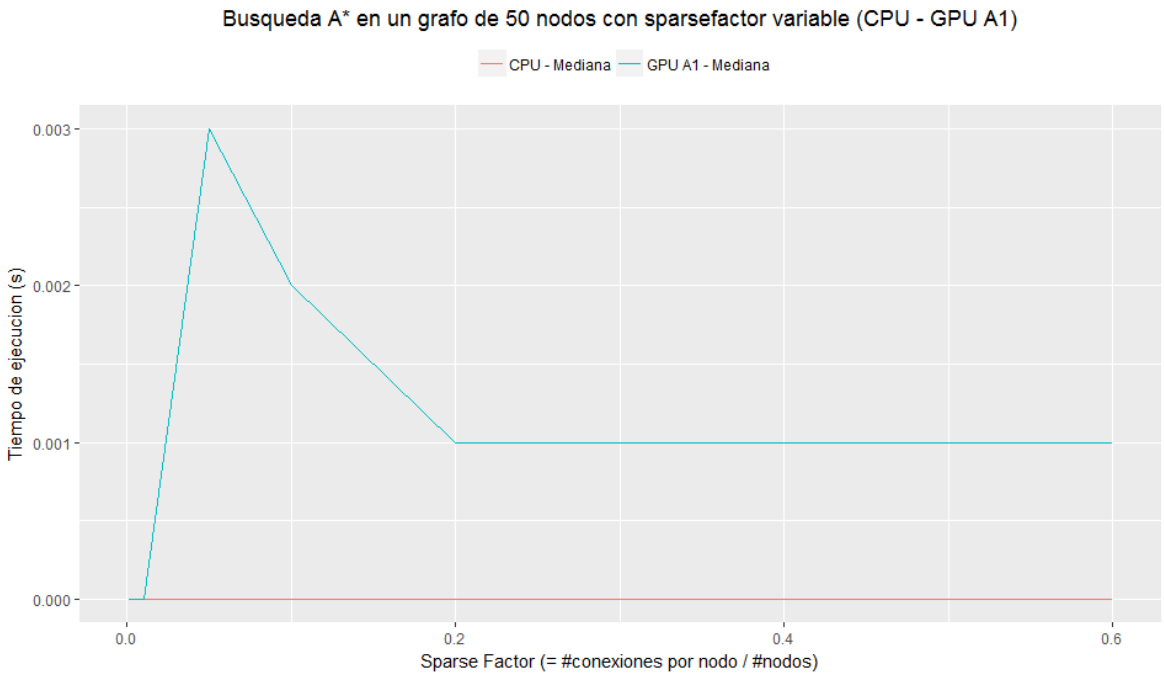

Figura 4-3 Prueba de la implementación A1 con 50 nodos

En esta primera imagen ya nos encontramos con un problema mencionado anteriormente: el *overhead* de las llamadas a las funciones de creación del kernel y búferes. Hemos tratado de minimizarlo, pero no es posible llegar a eliminarlo. Por tanto, las versiones de GPU del A* siempre empezarán con esa pequeña desventaja. También se puede ver otro inconveniente: un pico en las primeras ejecuciones que se debe a que la GPU no ha *calentado* (*warmup)*. Es un hecho común que debería poder solucionarse fácilmente ejecutando código en GPU antes de comenzar las mediciones. No obstante, ha resultado ser más problemático de lo esperado ya que no existe nada específico sobre cuánto debe durar dicha *warmup* y no siempre cumple su cometido.

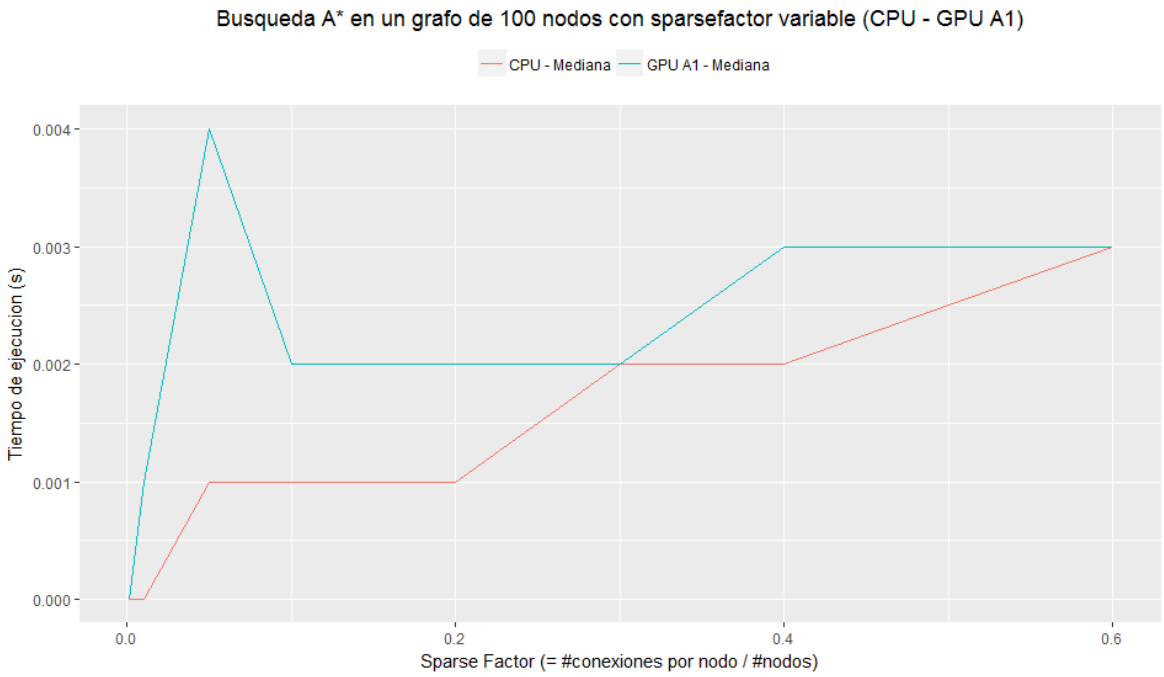

Figura 4-4 Prueba de la implementación A1 con 100 nodos

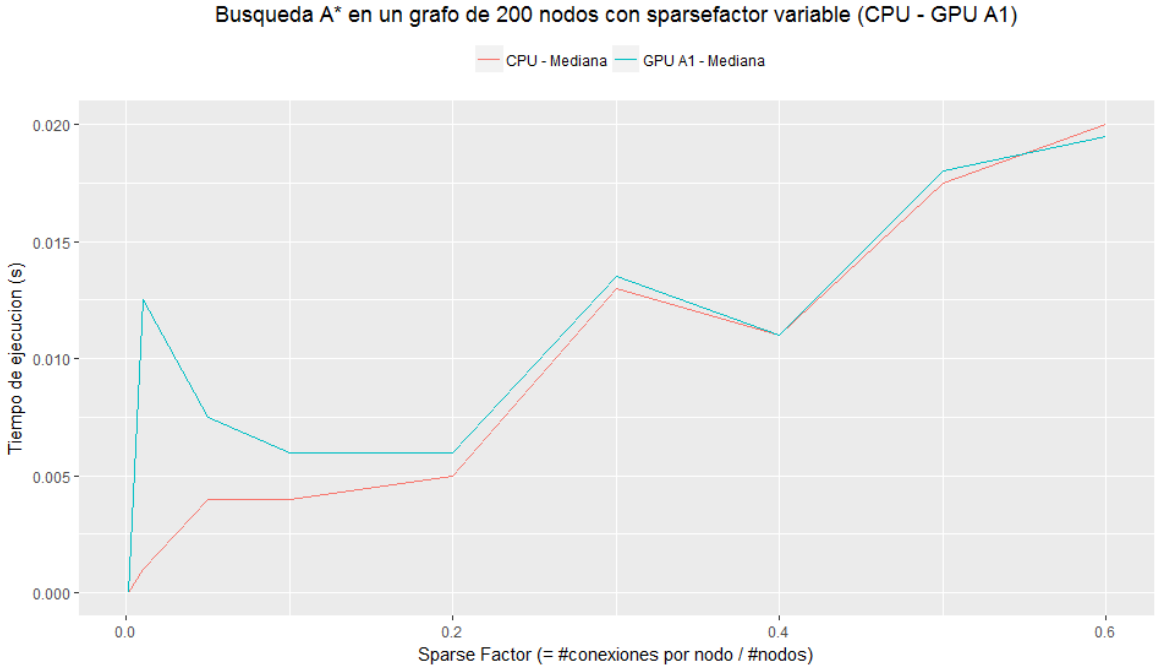

Figura 4-5 Prueba de la implementación A1 con 200 nodos

En la última vemos cómo se perfila el rendimiento del algoritmo creciendo de manera muy similar a la de CPU. Esto nos indica que paralelizar el cálculo heurístico no quita la suficiente carga de trabajo como para que sean notables las diferencias. Con una heurística computacionalmente compleja y pesada es posible que mejorase el rendimiento. Sin embargo, la heurística usada es demasiado simple (cálculo distancia euclidiana) y no puede representar ese caso.

4.2.2 Expansión de nodos en GPU: segunda implementación (A2)

Para nuestra segunda versión la GPU se encargará también de la búsqueda del costo *g* del sucesor. Es decir, buscará entre todas las aristas del grafo cual tiene como vértices el nodo en expansión y el nodo sucesor correspondiente.

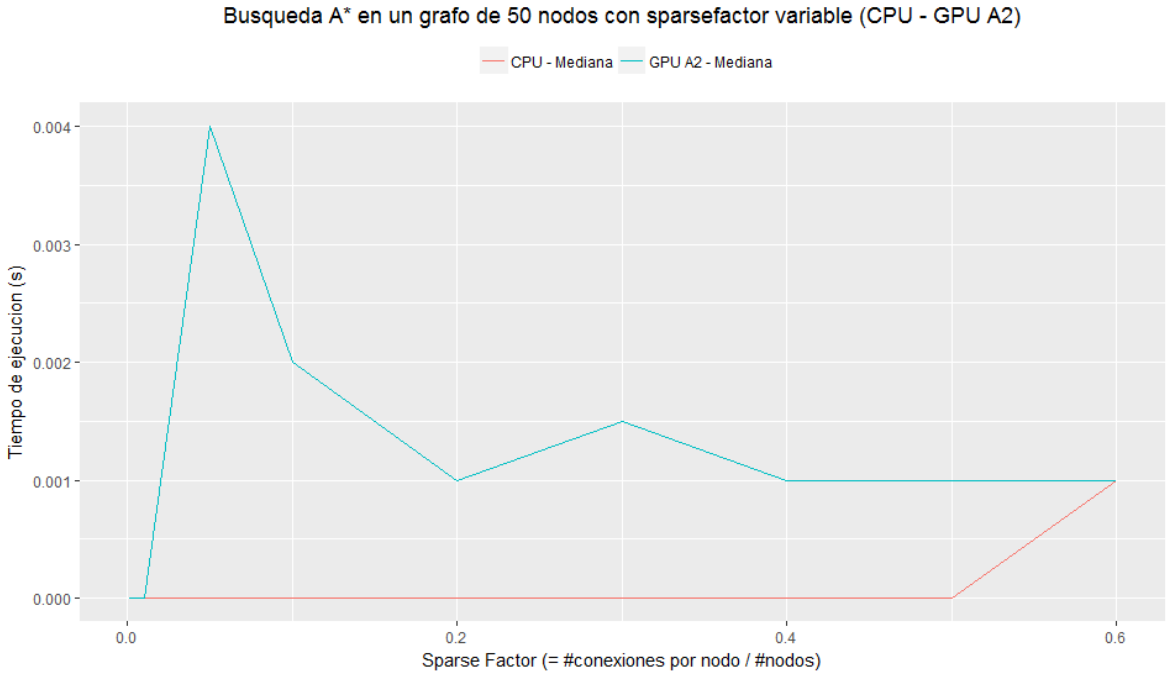

Figura 4-6 Prueba de la implementación A2 con 50 nodos

La diferencia entre la anterior implementación y ésta con grafos de 50 nodos es mínima.

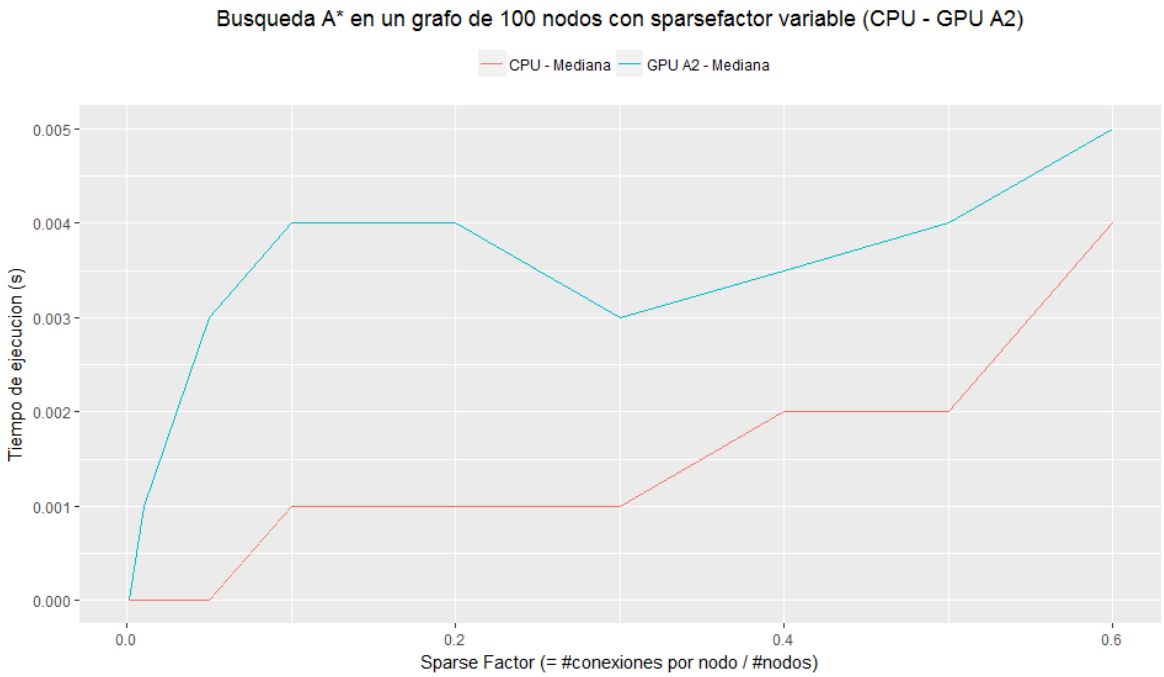

Figura 4-7 Prueba de la implementación A2 con 100 nodos

No obstante, a partir de 100 nodos observamos una diferencia muy notable.

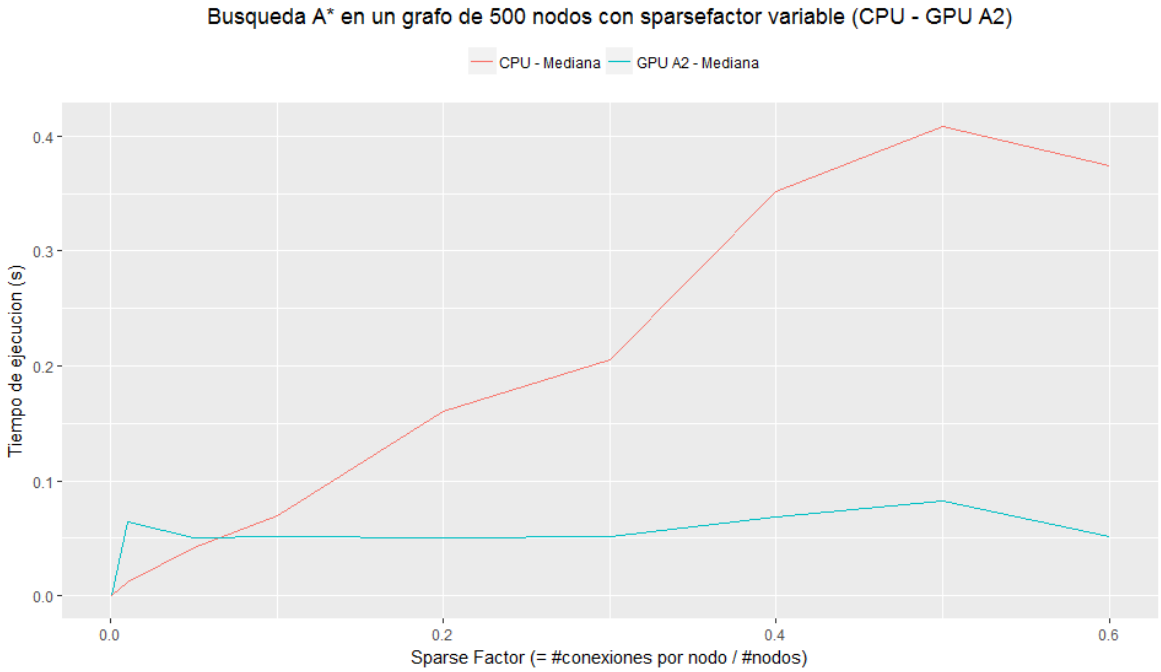

Figura 4-8 Prueba de la implementación A2 con 500 nodos

Con 500 nodos ya es clara la enorme diferencia que ha supuesto añadir esta búsqueda.

Podemos hipotetizar que el acceso a memoria tiene un alto coste temporal y produce un *cuello de botella* en CPU que se ve acrecentado a cuantos más nodos existan en el grafo (causa directa del aumento del valor de la densidad del grafo). Este hecho se ve reflejado en el crecimiento lineal de la CPU. Mientras tanto, la GPU no se ve afectada gracias a la paralelización del acceso a memoria.

4.2.3 Expansión de nodos en GPU: tercera implementación (A3)

En esta implementación ya optamos por una inserción completa de la expansión de nodos en GPU. Esto incluye la búsqueda en las listas de nodos abiertos y cerrados del nodo sucesor para valorar si es un posible candidato a expansión añadiéndolo a la lista de nodos abiertos.

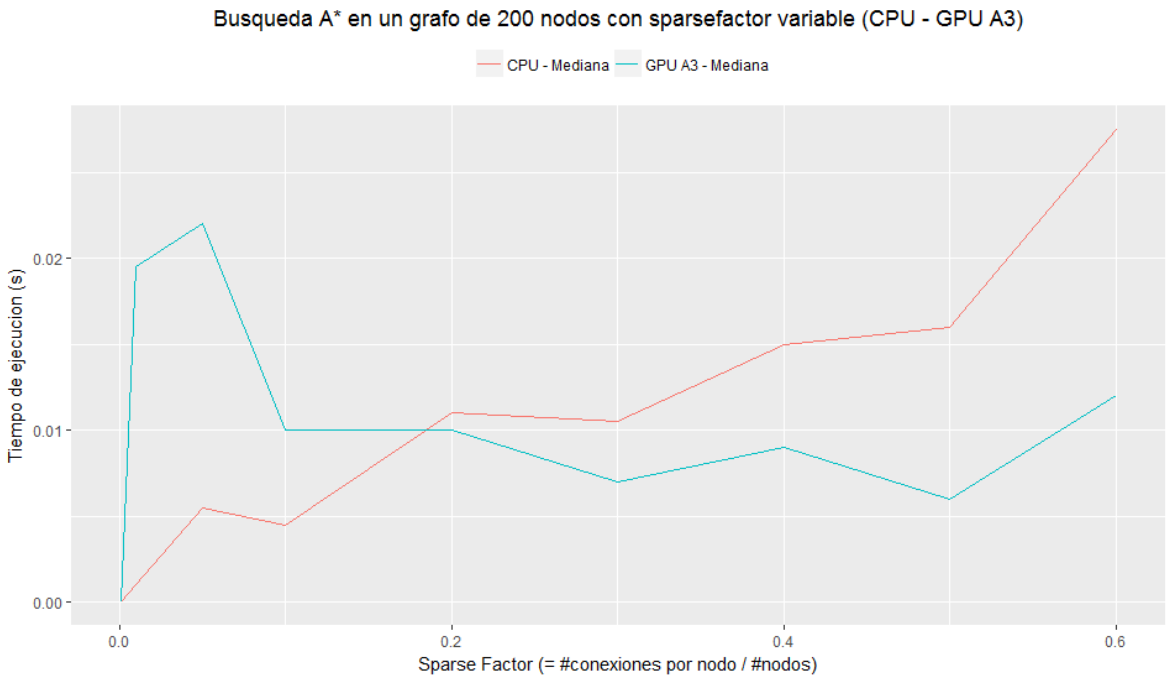

Figura 4-9 Prueba de la implementación A3 con 200 nodos

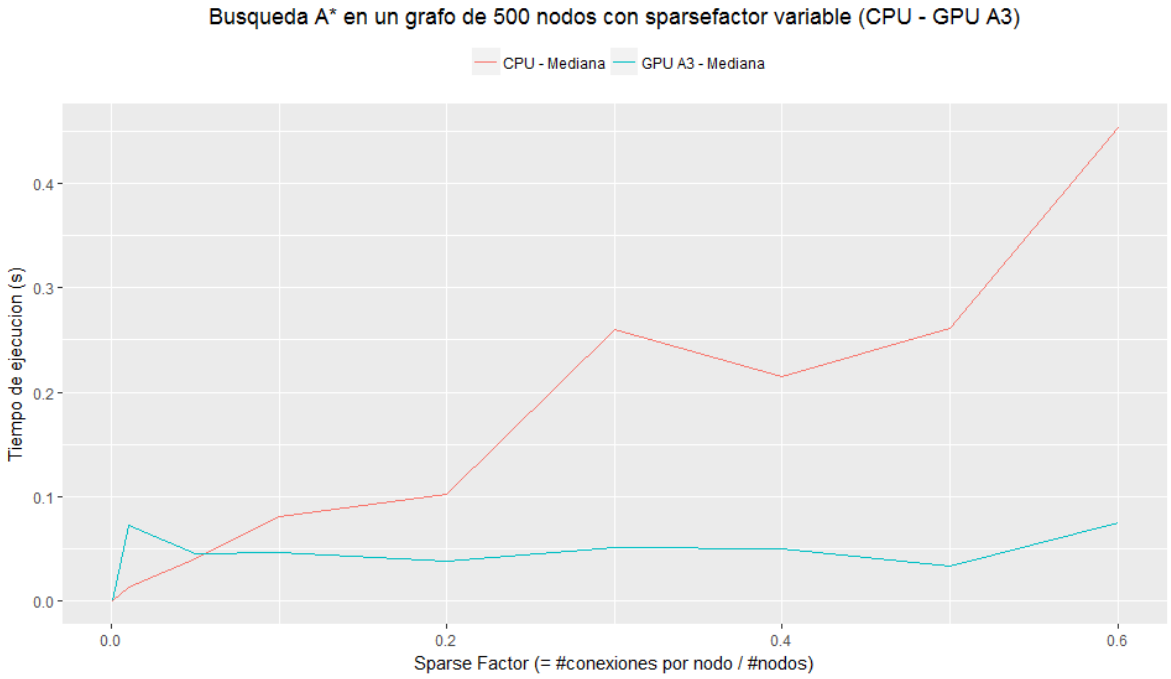

Figura 4-10 Prueba de la implementación A3 con 500 nodos

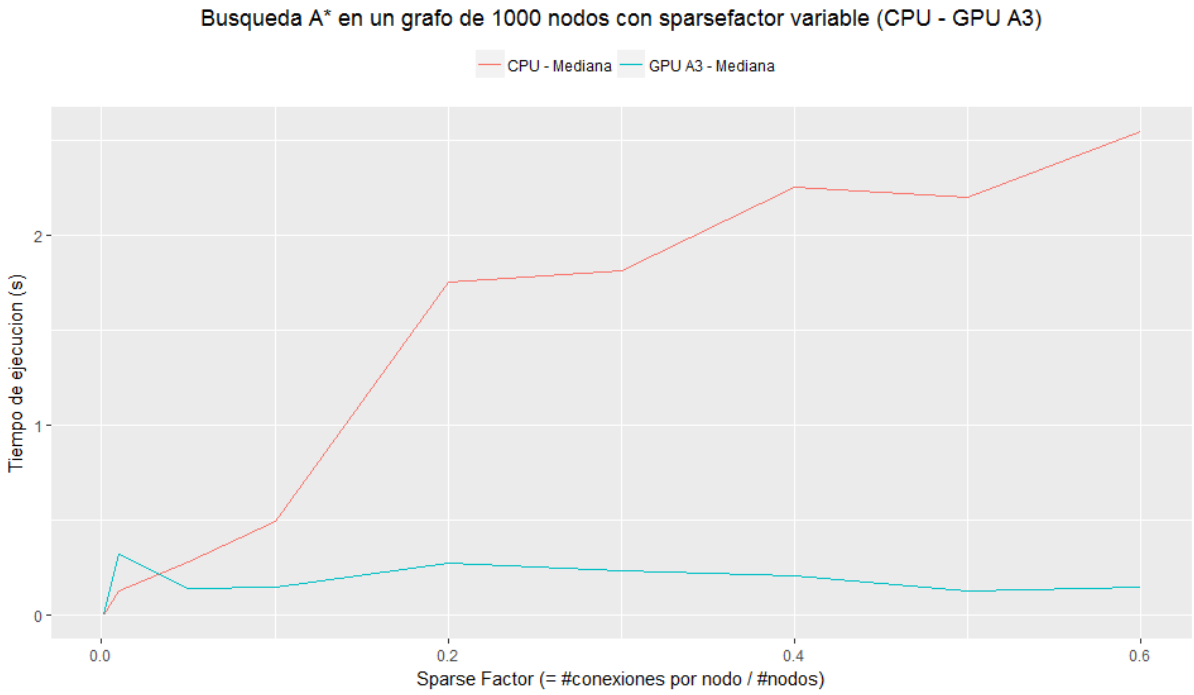

Figura 4-11 Prueba de la implementación A3 con 1000 nodos

A pesar de paralelizar sendas búsquedas, la diferencia con la implementación A2 es mucho menor que la que pudimos ver entre A1 y A2. Esto cobra sentido si nos fijamos en que la búsqueda del coste del orden O(|E|) mientras que las búsquedas en las listas son del orden O(|V|) donde

$$|E| = |V|*|V|*sparsefactor$$

Por tanto, a medida que aumenta *sparsefactor*, se incrementa considerablemente el costo temporal de las búsquedas del costo entre todas las aristas.

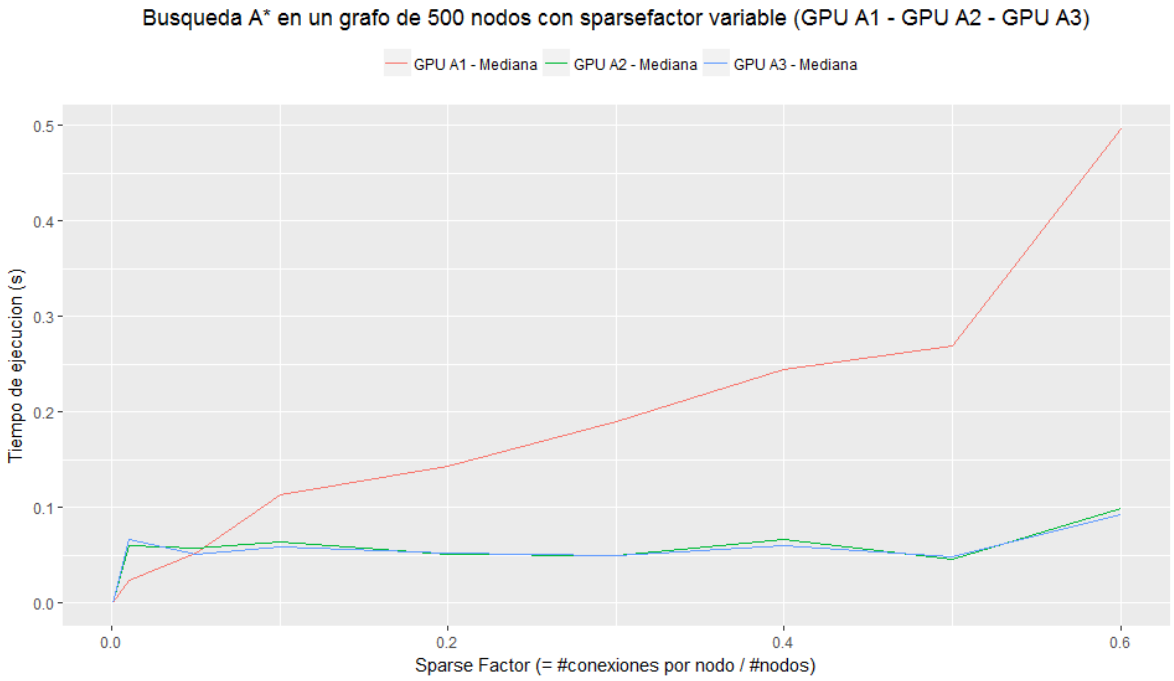

Figura 4-12 Implementaciones del grupo A

Ya sea con la segunda (A2) o tercera (A3) implementación, a partir de cierto número de nodos crítico, la eficiencia del A* con una parte paralelizada en GPU es estrictamente mejor que el algoritmo ejecutado únicamente en CPU. Viendo estos resultados, cabe preguntarse, ¿podríamos mejorar la eficiencia del algoritmo ejecutándose íntegramente en GPU? De esta manera evitaríamos tener que volver al *host* (CPU) y volver a cambiar búferes para insertar los nuevos datos. Por otro lado, perderíamos la alta frecuencia de reloj de la CPU y la parte más secuencial del algoritmo sufriría pérdidas de rendimiento al ser bastante menor la frecuencia de reloj de la GPU. A partir de estas preguntas surge el grupo B de implementaciones.

4.2.4 A* íntegro en GPU: expansión de nodos no paralelizada (B1)

A pesar de que ya intuimos qué resultados arrojará esta implementación, es interesante realizarla como paso intermedio y validación de nuestras hipótesis. Al realizar el algoritmo en un único hilo, su eficiencia debería ser bastante menor al de la CPU.

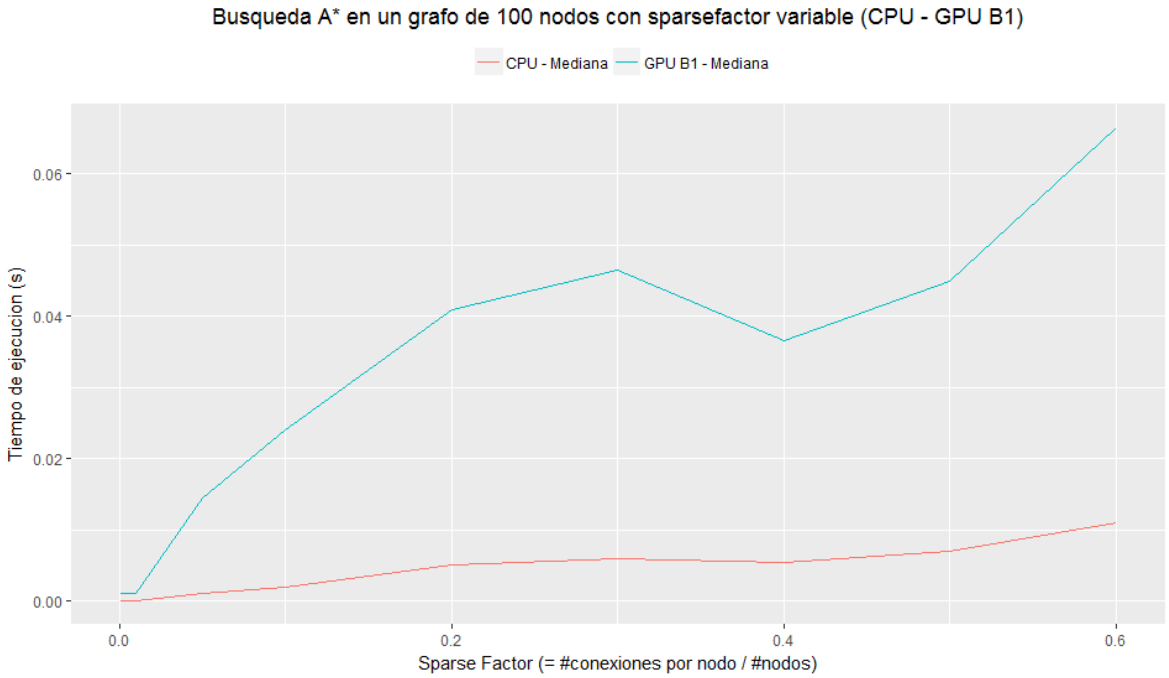

Figura 4-13 Prueba de la implementación B1 con 100 nodos

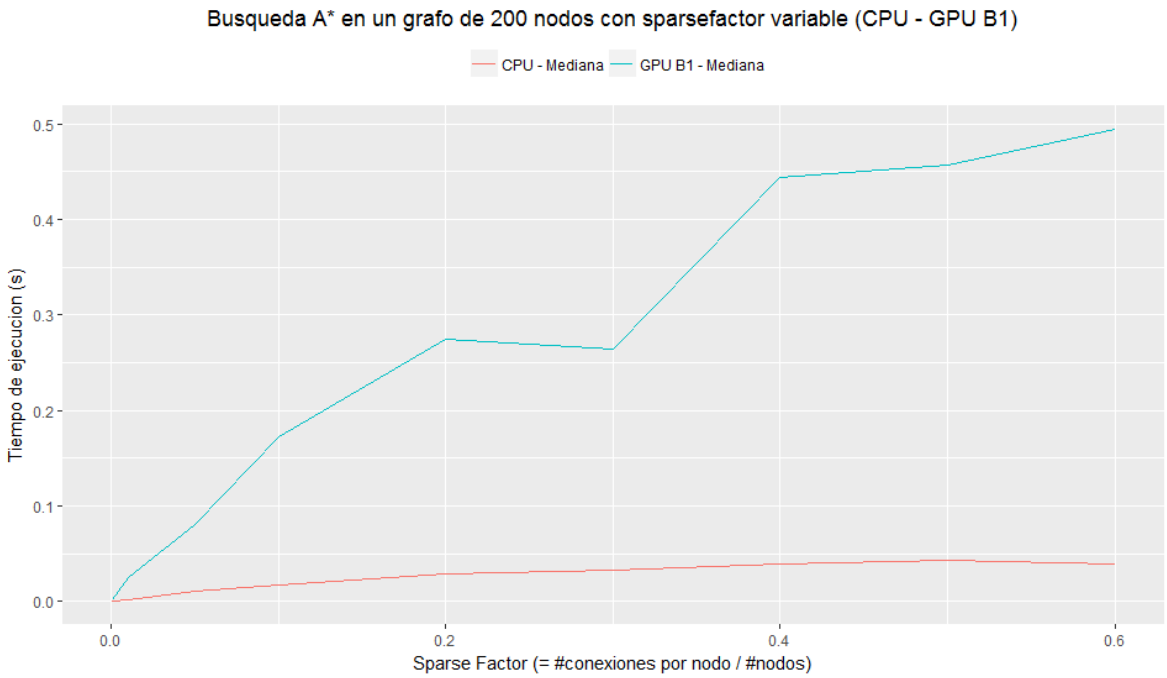

Figura 4-14 Prueba de la implementación B1 con 200 nodos

Los resultados confirman lo que ya intuimos: la ejecución de un mismo proceso en GPU es mucho más lento debido a la unidad de procesamiento más simple y la frecuencia de reloj menor.

4.2.5 A* íntegro en GPU: expansión de nodos paralelizada (B2)

En este ejemplo sí haremos uso de otros hilos para poder paralelizar la expansión de nodos. Es muy similar a la implementación A3 donde uno de nuestros hilos hará de *host* o hilo principal. Por desgracia, esta implementación no puede aprovechar todo el potencial de la GPU pues es necesaria la sincronización de varias variables entre los hilos y la especificación de OpenCL sólo exige sincronización entre hilos de un mismo un *WorkGroup*.

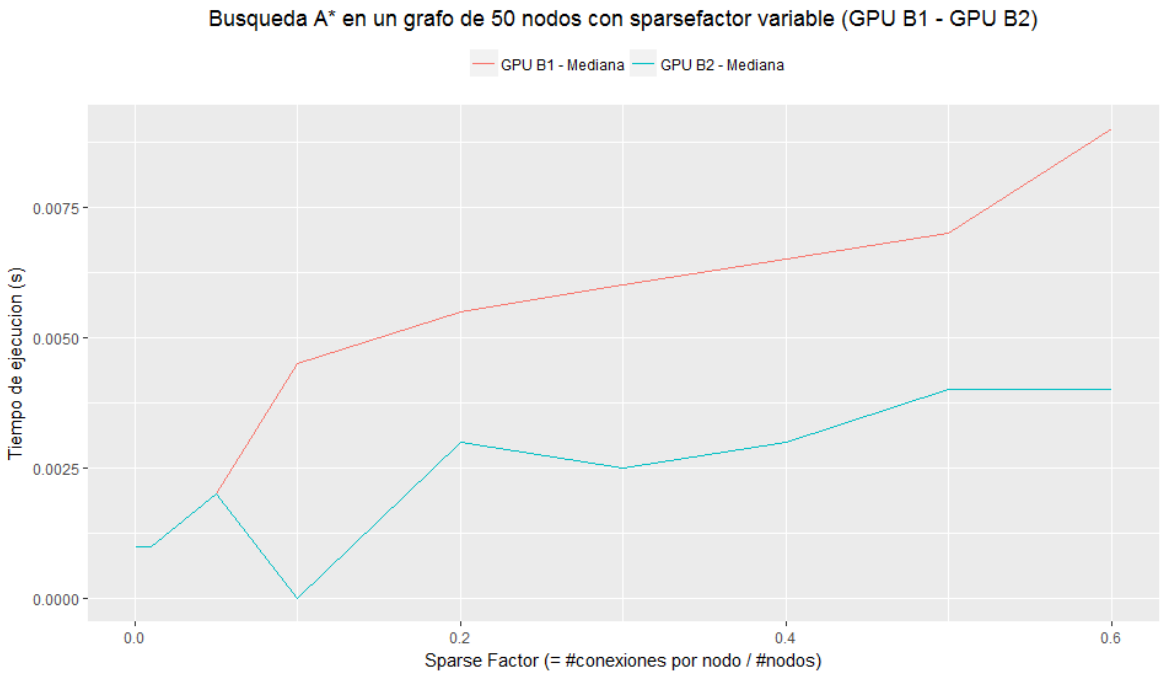

Figura 4-15 Comparación de B2 con B1 sobre un grafo de 50 nodos.

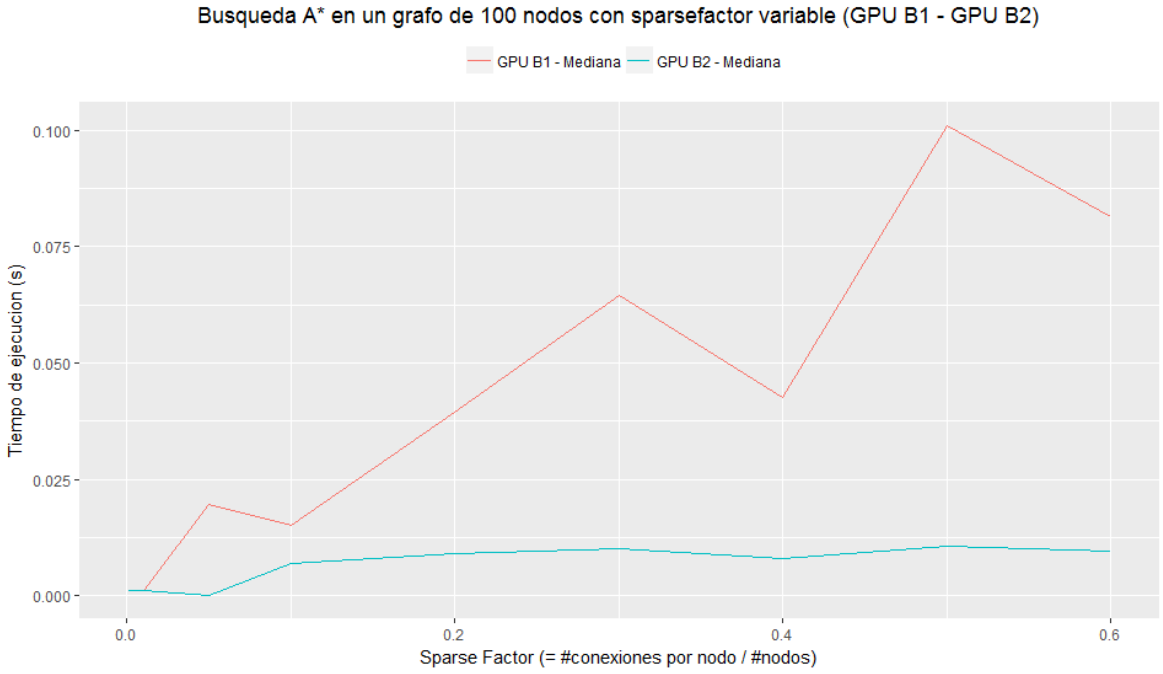

Figura 4-16 Comparación de B2 con B1 sobre un grafo de 100 nodos.

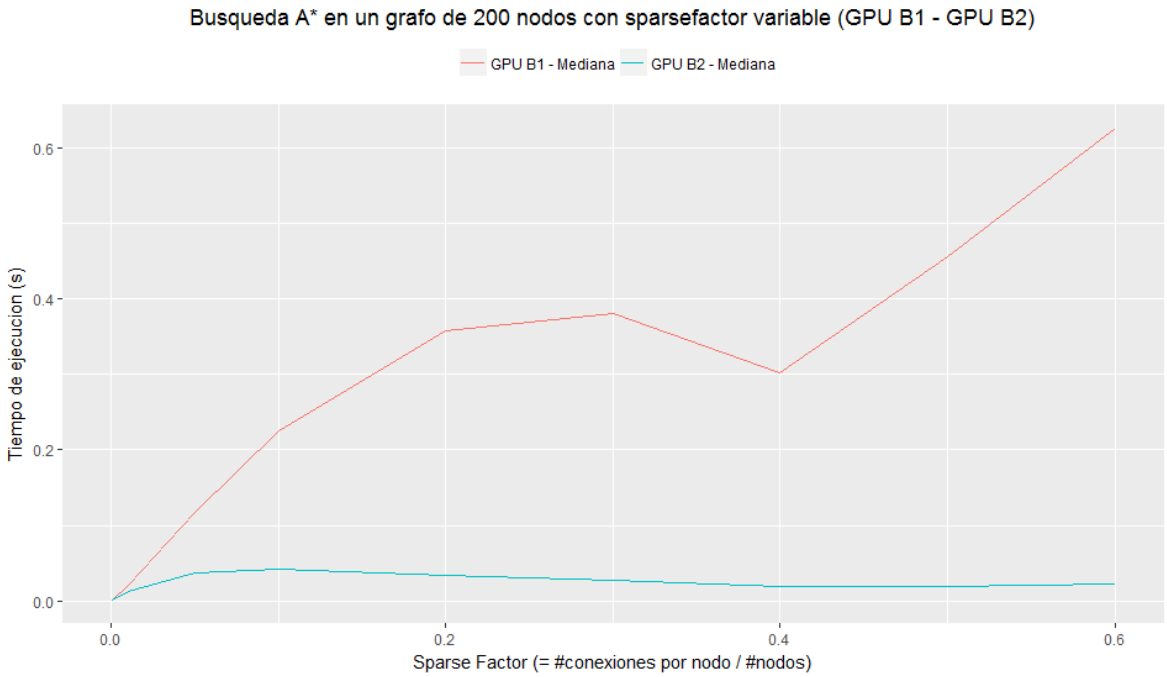

Figura 4-17 Comparación de B2 con B1 sobre un grafo de 200 nodos.

Conseguimos una mejora de rendimiento similar a la que sucedió entre la implementación en CPU y A2. Para comprobar verdaderamente si es igual de buena que A3, comparémoslas:

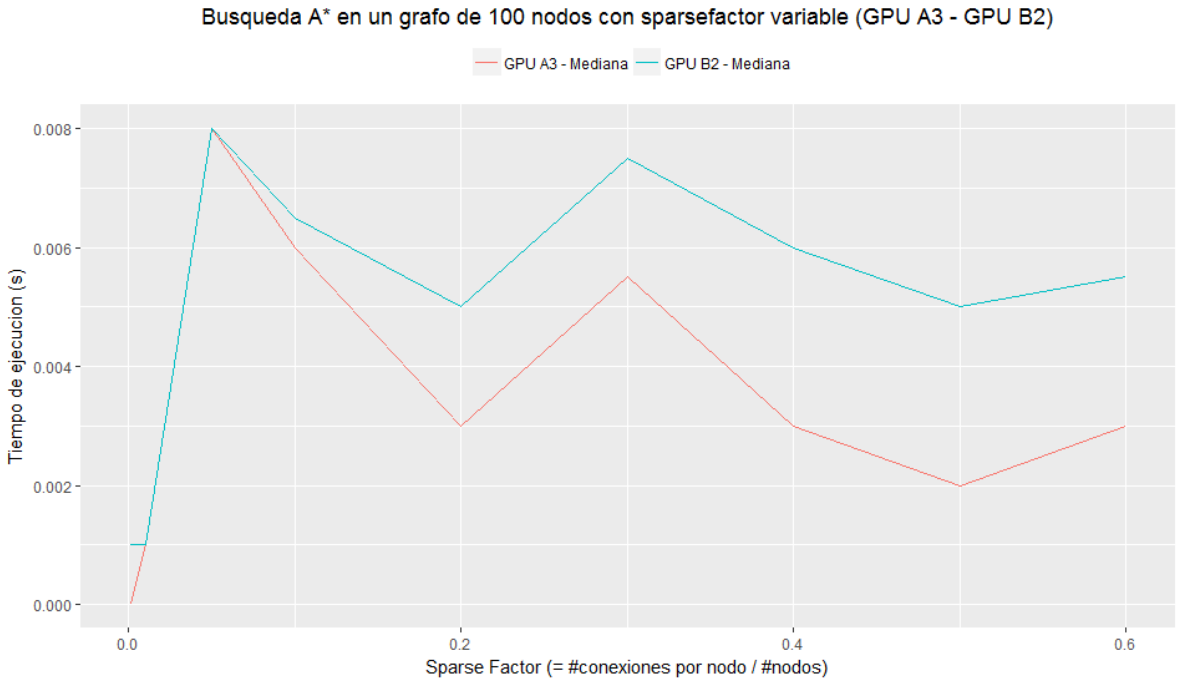

Figura 4-18 Comparación de B2 con A3 sobre un grafo de 100 nodos.

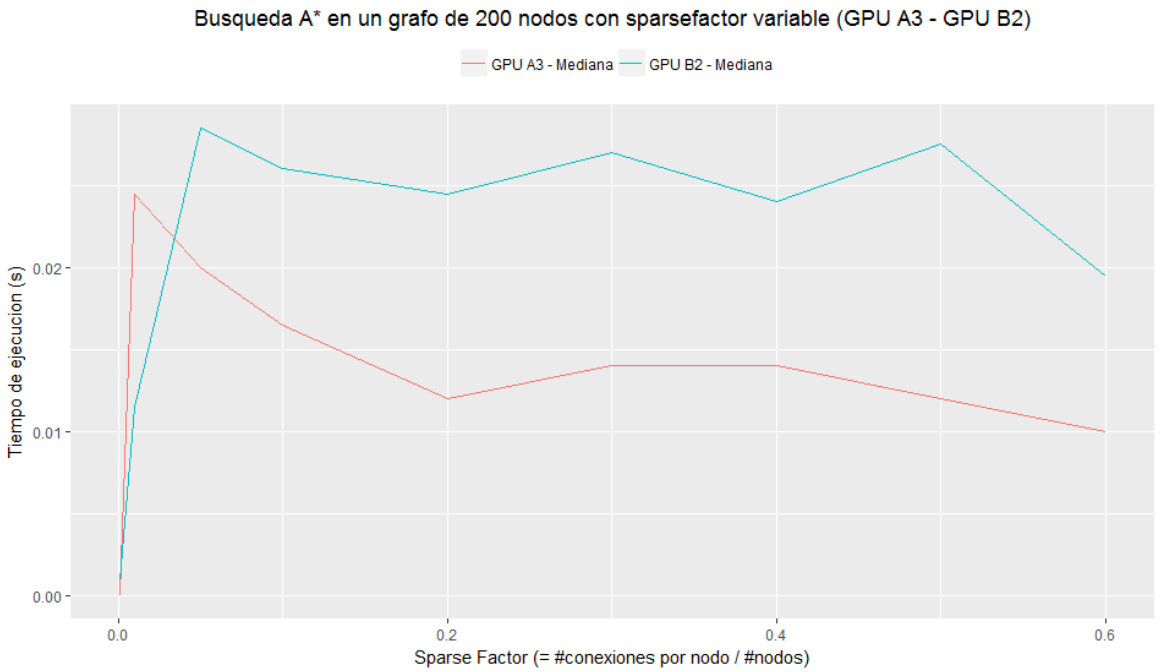

Figura 4-19 Comparación de B2 con A3 sobre un grafo de 200 nodos.

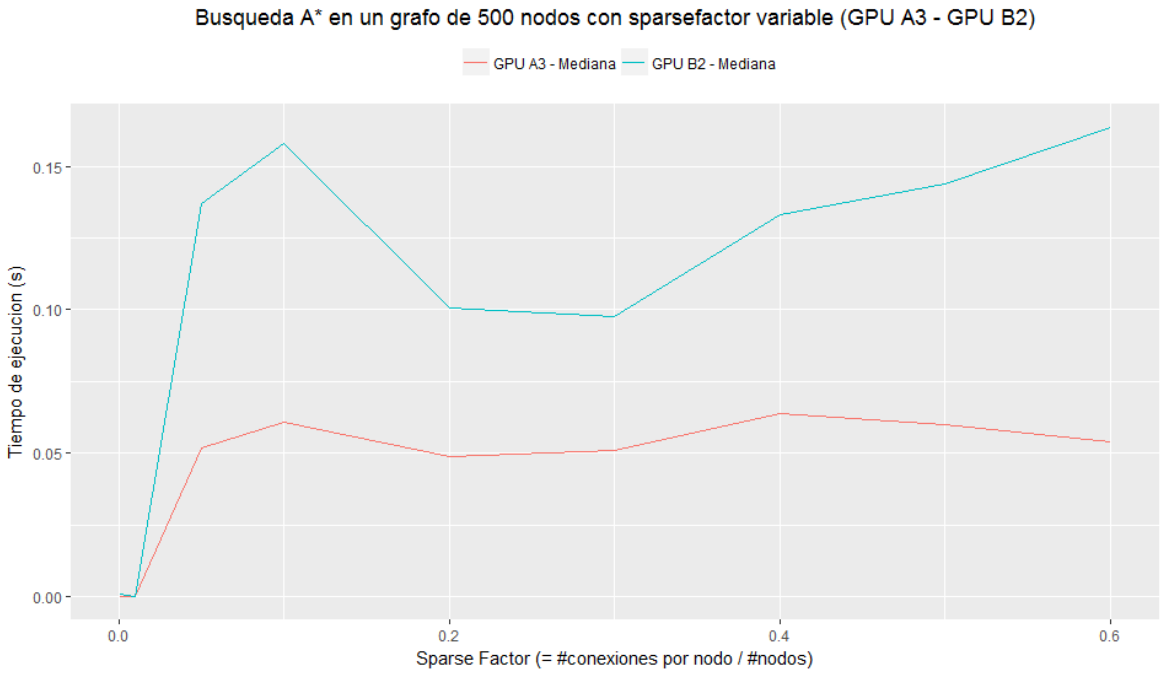

Figura 4-20 Comparación de B2 con A3 sobre un grafo de 500 nodos.

A la vista de estos resultados, podemos concluir que una implementación íntegra en GPU no es óptima. Es posible que se deba a que el hilo principal hace *cuello de botella* a esta implementación al no ir tan rápido como iría la CPU ejecutando las mismas instrucciones. Para obtener una implementación rápida y eficiente es vital discernir correctamente entre las partes del algoritmo que van a aprovechar el potencial de la CPU (secuencial, sincronización) y aquellas que se benefician de una ejecución en paralelo en la GPU (independencia de variables, alto número de hilos).

En ciertos géneros de videojuegos es corriente tener cientos de IAs en activo. Es común también que estos juegos se encuentren limitados por la CPU. Por ello, sería muy interesante ejecutar los cálculos de esas IAs en GPU. Exploramos esta posibilidad en la última implementación.

4.2.6 Múltiples instancias en CPU y GPU (C)

Para poder ejecutar varias instancias de A* en GPU haremos uso de B1 y llamaremos a varios hilos. La razón de no usar B2 (que vimos era más eficiente) es que la implementación hace uso de todo un *WorkGroup* y, por un lado, el número de instancias que podríamos ejecutar en paralelo estaría muy limitado y, por el otro, habría que recodificar la implementación para que no existieran conflictos al llamar a más instancias. En CPU simplemente llamaremos secuencialmente a las demás instancias.

Probemos con un número fijo de 500 instancias:

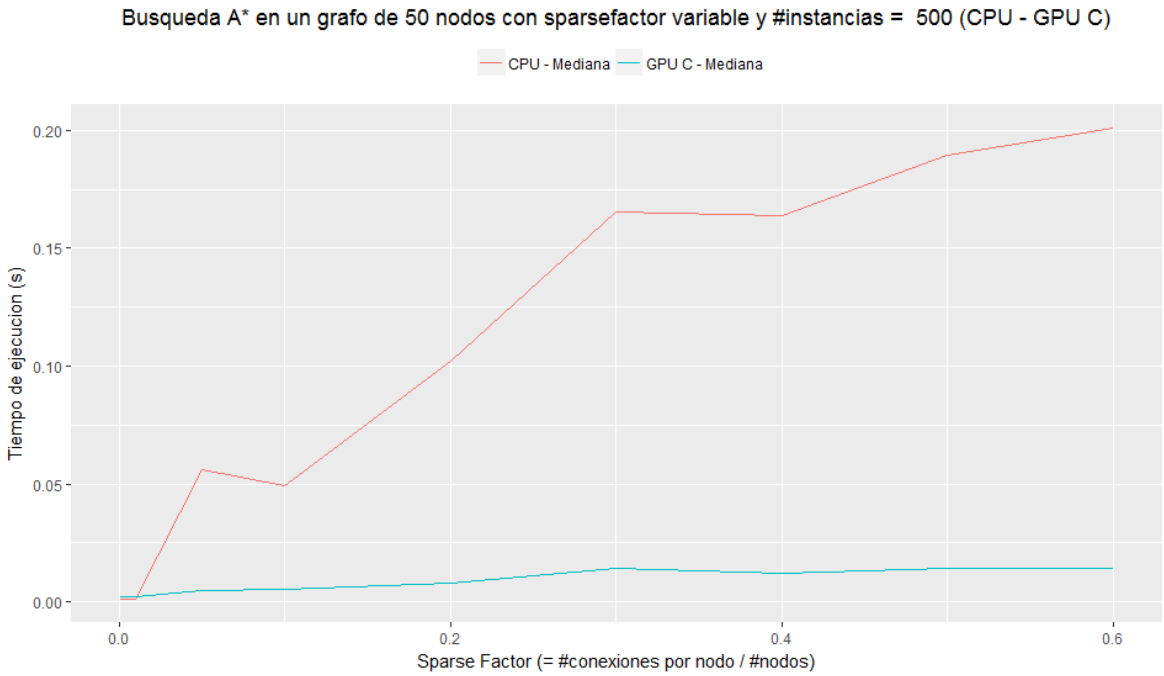

Figura 4-21 Ejecución de 500 instancias de las implementaciones CPU y B1 en un grafo de 50 nodos.

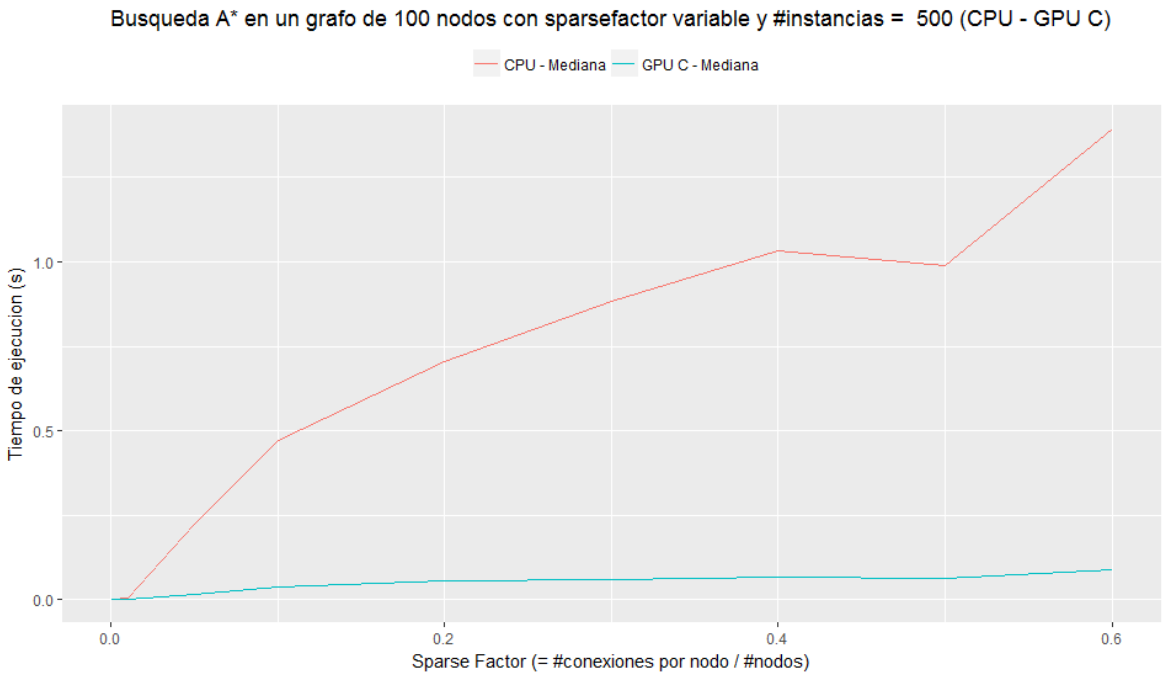

Figura 4-22 Ejecución de 500 instancias de las implementaciones CPU y B1 en un grafo de 100 nodos.

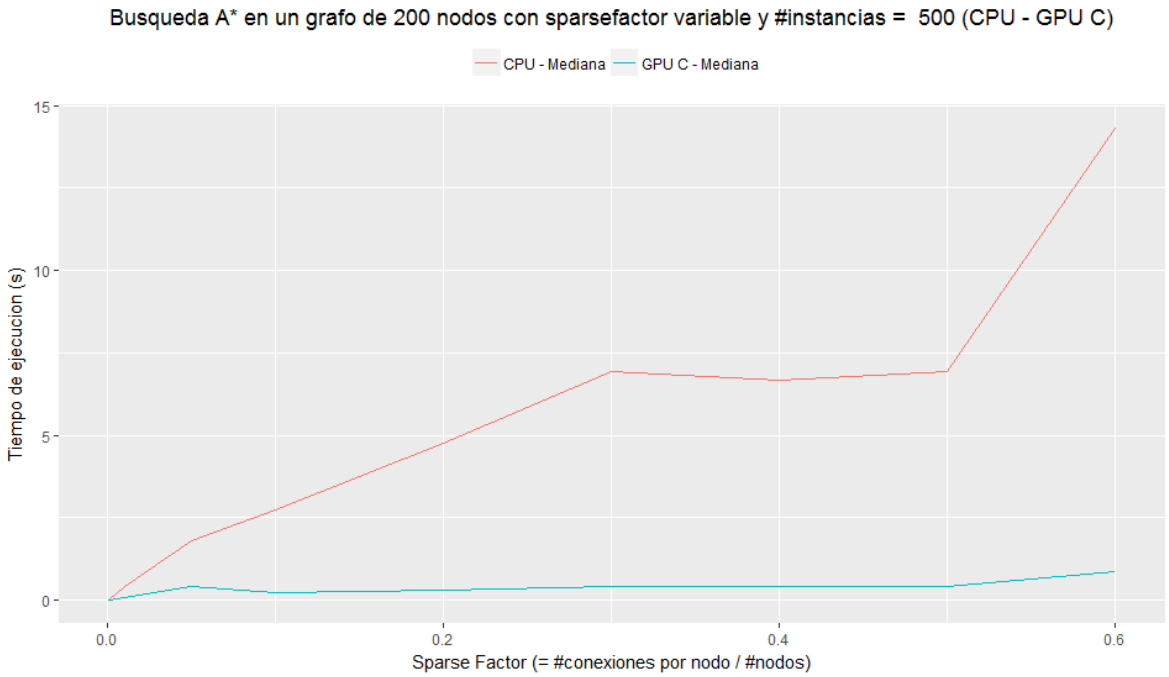

Figura 4-23 Ejecución de 500 instancias de las implementaciones CPU y B1 en un grafo de 200 nodos.

Son resultados bastante prometedores que nos indican la potencia de llevar a la vez varias ejecuciones de un algoritmo en GPU. Veamos con las siguiente gráficas valores menores para el número de instancias:

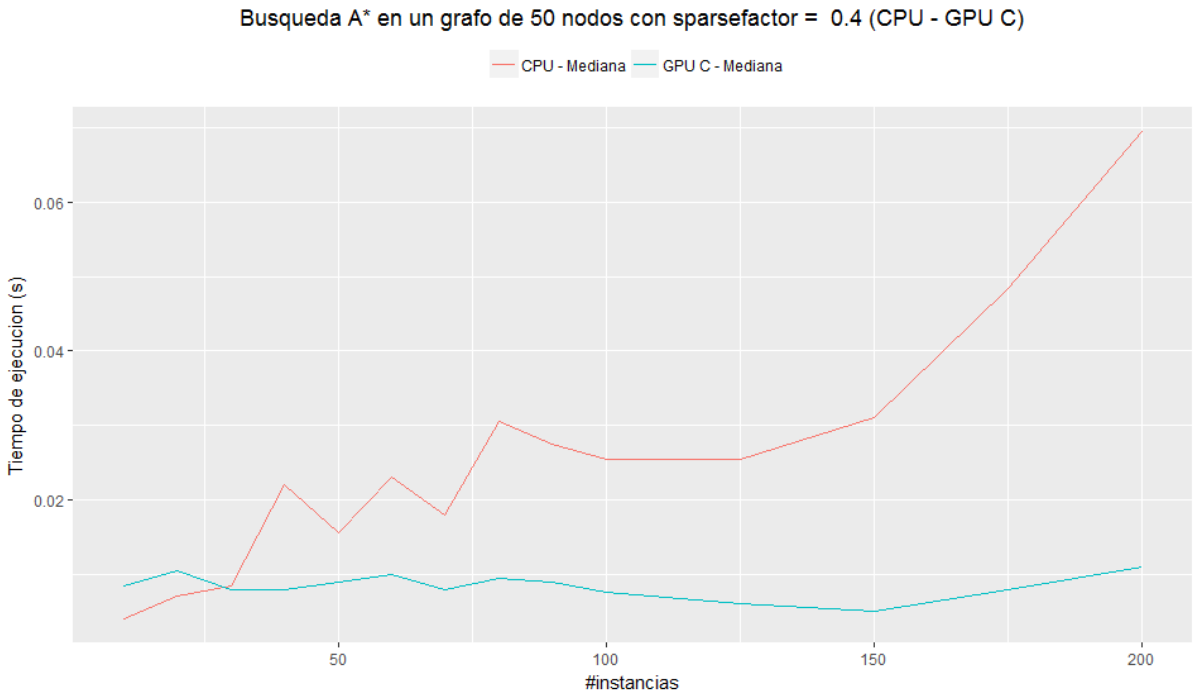

Figura 4-24 Comparación de CPU y B1 en un grafo de 50 nodos y sparsefactor = 0.4.

Aún con un valor bajo del número de nodos, alrededor de las 30 instancias, la GPU empieza a ser más eficiente.

5 Conclusiones y trabajo futuro

5.1 Conclusiones

A lo largo de la presentación de este proyecto hemos podido vislumbrar el potencial de las GPUs y cómo pueden ser de ayuda en tareas que, antes pensábamos, sólo estaban destinadas para la CPU. Aunque si dichas tareas son pequeñas, el uso de una GPU puede resultar perjudicial para el rendimiento.

5.1.1 Aspectos negativos

Primer problema de todos y crucial: los videojuegos necesitan tiempos de respuesta excesivamente rápidos. Los desarrolladores deben optimizar sus juegos para conseguir que la GPU dibuje, como mínimo, 30 veces por segundo por pantalla el estado del videojuego. Esto significa que tienen alrededor de 0.033 segundos (o 33 ms) para realizar todos los cálculos relacionados con gráficos, lógica, IA… Esto ha ocasionado que los desarrolladores expriman al máximo la capacidad computacional de las GPU usando distintos métodos para reducir cálculos. Desafortunadamente, estos métodos afectan a la IA haciéndola más básica para consumir menos recursos. Y por ello, es muy posible que una implementación en GPU de sus IAs no mejorase el rendimiento porque están codificadas para reutilizar gran parte de sus resultados.

Otra dificultad encontrada ha sido el lenguaje de programación. OpenCL C es muy parecido a C, uno de los lenguajes más cercanos a lenguaje máquina y eficientes por esta misma razón. Pero están ausentes dos características principales: el uso de punteros a memoria y la creación dinámica de búferes. La falta de estas herramientas puede producir que el algoritmo implementado sea menos eficiente. Además, implica retos adicionales que muchos desarrolladores no están dispuestos a asumir.

5.1.2 Aspectos positivos

Los aspectos positivos están estrechamente relacionados con los puntos negativos anteriores. Creemos que los argumentos anteriores son negativos únicamente debido a que actualmente la programación en paralelo no está tan extendida como la secuencial y no se dan a conocer todos sus beneficios por lo que no existe incentivo para usarla. Si no fuera así el caso y más programadores la usasen y explorasen con ella, surgirían métodos para reducir cálculos igual que en CPU y existiría más información y documentación para crear algoritmos en GPU.

Por otro lado, hemos podido observar como una implementación híbrida (parte en CPU, parte en GPU) los resultados son muy satisfactorios. A medida que aumenta el tamaño de un grafo, la implementación en GPU sufre mucho menos en rendimiento en comparación con la implementación usual en CPU. Esto abre la posibilidad de usar la GPU para este tipo de cálculos con la confianza de que se obtendrán buenos resultados.

5.2 Trabajo futuro

- Seleccionar más algoritmos para su estudio. Existen múltiples algoritmos comunes usados para Inteligencia Artificial de juegos. Por ejemplo, el algoritmo Minimax (con o sin poda alfa-beta) es un candidato perfecto para implementar en paralelo pues evalúa estados del tablero con independencia de sus nodos hermanos.

- Insertar en un videojuego real esta implementación y hacer varias mediciones para comprobar si consigue mejorar su rendimiento. Los juegos ideales, y donde el nuevo algoritmo podría destacar más, son aquellos pertenecientes al género RTS (*Real Time Strategy*) ya que hay que calcular de manera constante y rápida los caminos que seguirán centenares de unidades.

- Sin duda no es una implementación perfecta. Se puede refinar su acceso a memoria o intentar reducir los parámetros que se le pasan para minimizar el *overhead* de la llamada al *kernel*. Con esto se podría intentar mejorar los resultados para grafos con una cantidad pequeña de nodos donde la GPU, ahora mismo, es menos eficiente.

- No se han podido realizar pruebas extensivas en hardware diferente. En nuestro caso ha sido una combinación de una CPU más antigua que la GPU. pero sería interesante probar otras combinaciones y marcas.

Referencias

[1] Rafia Inam, “A* Algorithm for Multicore Graphics Processors”, Master’s Thesis, Chalmers University of Technology, Göterborg 2009. 62p. http://publications.lib.chalmers.se/records/fulltext/129175.pdf

[2] Yichao Zhou, Jianyang Zeng, “Massively Parallel A* Search on a GPU”, Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, Tsinghua University, Beijing. 7p. https://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/download/9620/9366.

[3] “A* search algorithm”, Wikipedia, https://en.wikipedia.org/wiki/A*_search_algorithm

[4] Rajiv Eranki, “Pathfinding using A* (A-Star)”, 2002 http://web.mit.edu/eranki/www/tutorials/search/

[5] “Graphics processing unit”, Wikipedia, https://en.wikipedia.org/wiki/Graphics_processing_unit

[6] “Who rules entertainment industry? Video games vs. film & music”, Rob Foote, http://www.kfvs12.com/story/31140647/who-rules-the-entertainment-industry

[7] Simon Kemp, We are Social, https://wearesocial.com/uk/special-reports/digital-in-2017-global-overview

[8] https://developer.nvidia.com/deep-learning

[9] “Inceptionism: Going Deeper into Neural Networks”, Alexander Mordvintsev, Christopher Olah, Mike Tyka, Google Research, 2015, https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html

[10] “General-Purpose computing on Graphics Processing Units”, Wikipedia, https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units

[11] OpenCL specification, Khronos Group, https://www.khronos.org/opencl/

[12] OpenCL 1.1 documentation, https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/

[13] FLOPS de AMD CPU Phenom II x4 970, http://www.overclock.net/t/869018/intel-and-amd-gflops-data-thread-looking-for-sandybridge-too

[14] FLOPS de AMD GPU R9 Fury, http://wccftech.com/review/sapphire-r9-fury-nitro-review/

[15] Diferentes cuestiones durante el desarrollo de las implementaciones, https://stackoverflow.com/

Glosario

GPU	Graphics Processor Unit.
CPU	Central Processing Unit.
GPGPU	General-Purpose Computing on Graphics Processing Units.
IA	Inteligencia Artificial.
Kernel	Función principal del código a ejecutar en GPU.
Flop	Floating Point Operation per second, usada para medir la potencia de una unidad de procesamiento.
Buffer	Array de datos de un mismo tipo que se pasa como parámetro a un kernel.
CommandQueue	Objeto de OpenCL encargado de ejecutar comandos para la GPU.
Pathfinding	Búsqueda de caminos. Término utilizado en grafos, mapas…
WorkGroup	Grupo de hilos ejecutados concurrentemente en GPU en una unidad de computación (CU). Existe sincronización de datos entre los hilos.

Anexos

A Implementaciones en detalle

En esta sección explicaremos con un poco más de detalle, adjuntado código fuente, las distintas partes del código pertenecientes a la ejecución en GPU, es decir, los *kernel*. No nos detendremos a mostrar la parte en CPU ya que lo único interesante o novedoso dentro de esa parte es el orden de llamadas de las funciones de OpenCL para crear los *kernel*. Sin embargo, esa parte ya fue vista en el apartado 3.3.

Aunque no sea expuesto a continuación, todo el código fuente referente al proyecto se puede encontrar en el siguiente repositorio: https://github.com/bobemv/a-search-gpu

Estructuras

Las siguientes estructuras han sido necesarias para mantener una sincronización de los datos de la búsqueda A* entre el *host* y el *device*:

```
typedef struct infonode {
    ulong id;
    float x;
    float y;
}infonode;

typedef struct node {
    ulong id;
    ulong parent;
    ulong type;
    float f;
    float g;
    float h;
}node;

typedef struct edge {
    ulong from;
    ulong to;
    ulong cost;
}edge;
```

Caben destacar de ellas dos cosas:

- La estructura del nodo (*node*) del grafo es ligeramente diferente a la de una implementación corriente. Posee un identificador que está relacionado con el hecho de que los nodos padres son referenciados mediante dicho campo y no con un puntero directamente. Esto se debe a una de las limitaciones de OpenCL: no es posible la creación de punteros. Para tener unas implementaciones lo más parecidas posibles, la de CPU también usa esta estructura, aun cuando podría usar otra.

- El grafo está compuesto por varias instancias de *edge*. Al igual que con el punto anterior, esto se debe a la limitación de no poder tener objetos de tamaño dinámico dentro de una estructura. Si no existiera, podría haber sido usado una estructura más parecida a *map*, con todas las ventajas que acarrea.

Hemos decidido usar *unsigned long* (*ulong* en OpenCL) en vez de *int* para poder utilizar el algoritmo con grafos de un tamaño aún mayor. Aunque para nuestros casos, *int* habría sido más que suficiente.

Funciones adicionales

Las funciones adicionales son:

```
float heuristic(__global infonode *infonodes, const ulong idStart,
const ulong idEnd)
```

Devuelve la estimación del coste para llegar desde un nodo actual al nodo final. En nuestro caso, hemos usado una heurística sencilla que calcula la distancia euclidiana entre sendos nodos (cada nodo tiene una información adicional que los sitúa sobre un espacio bidimensional con coordenadas X e Y).

```
ulong search_cost_node_2_node(__global edge *conexiones, ulong nedges,
ulong from, ulong to)
```

Busca en el grafo el coste del camino entre dos nodos. La búsqueda es secuencial.

```
ulong genera_sucesores(__global node *sucesores, __global edge
*conexiones, const node nodo, const ulong nedges, const ulong
indexnodes)
```

Genera los sucesores de un nodo (busca aquellos nodos hijo que se pueden llegar directamente desde el nodo padre).

```
void bubblesort(__global node *elems, const ulong numElems)
```

Implementación estándar del algoritmo de ordenación *bubblesort* con flag de parada si la lista de elementos ya está ordenada. No ha podido ser usados algoritmos más rápidos como *MergeSort*, *QuickSort* o *HeapSort* debido a que la recursión no está permitida en OpenCL.

Kernel A1

```
// Implementación A1: paralelización del cálculo de la heurística
// en la expansión de nodos de la búsqueda A*
__kernel void searchastar(__global infonode *infonodes,
                          __global node *nodes,
                          const ulong nsucesores,
                          const ulong idEnd,
                          const ulong nnodos){
    int num = get_global_id(0);
    if(num >= nsucesores){
        num = 0;
    }
    nodes[num].h = heuristic(infonodes, nodes[num].type, idEnd);
}
```

El *kernel* de esta implementación es muy sencillo. Le proveemos con los nodos sucesores e información adicional necesaria. El *kernel* trabajará sobre el mismo búfer que se le pasa por parámetros y que luego el *host* se encargará de leer para obtener los resultados. Puesto que es posible que hayamos iniciado más hilos de los que necesitamos (es recomendable iniciar un número de hilos múltiplo de un valor que varía entre dispositivos), comprobamos que su identificador no sea mayor que la cantidad de sucesores que tenemos. Si es así, el hilo repite el mismo proceso que el hilo con identificador igual a 0.

Los búferes que usaremos a lo largo de las implementaciones son de tipo *__global*. Existen los tipos *__local* y *__private* también, no obstante, con *__global* los datos del búfer son compartidos entre todos los hilos de la GPU (con el inconveniente de que el tiempo de acceso a él es mayor).

Kernel A2

```
// Implementación A2: paralelización del cálculo de la heurística
// y búsqueda del coste nodo a nodo en la expansión de nodos de la
búsqueda A*
__kernel void searchastar(__global infonode *infonodes,
                          __global edge *conexiones,
                          __global node *nodes,
                          const node actual,
                          const ulong nsucesores,
                          const ulong idEnd,
                          const ulong nnodos,
                          const ulong nedges){
    int num = get_global_id(0);
    if(num >= nsucesores){
        num = 0;
    }
    nodes[num].h = heuristic(infonodes, nodes[num].type, idEnd);
    nodes[num].g = actual.g + search_cost_node_2_node(conexiones,
nedges, actual.type, nodes[num].type);
    nodes[num].f = nodes[num].g + nodes[num].h;

}
```

Muy parecida a la anterior implementación. Con un mayor número de parámetros porque requiere más información.

Kernel A3

```
// Implementación A3: paralelización de la expansión de nodos
// en el algoritmo A*
__kernel void searchastar(__global infonode *infonodes,
                          __global edge *conexiones,
                          __global node *abiertos,
                          __global node *cerrados,
                          __global node *sucesores,
                          __global ulong *out,
                          const node actual,
                          const ulong nsucesores,
                          const ulong nabiertos,
                          const ulong ncerrados,
                          const ulong nnodos,
                          const ulong nedges,
                          const ulong idEnd)
```

En esta implementación le proveemos con toda la información de la búsqueda A*. Tenemos un búfer, *out*, que es únicamente de salida. De tamaño *nsucesores*, con él podremos dar información adicional al *host* sobre los nodos sucesores que, de otra manera, no podríamos dar mediante los otros búferes. Cada *ulong* hace referencia a un nodo sucesor (el que está en su misma posición en su búfer). Tiene tres estados:

- 0: El nodo no es candidato a ser expandido.
- 1: El nodo sí es candidato a ser expandido.
- 2: Es el nodo final y se debe detener la búsqueda.

```
{
    int num = get_global_id(0);
    int i, j;
    node sucesor;
    if(num >= nsucesores){
        num = 0;
    }
    sucesor = sucesores[num];
    if (sucesor.type == idEnd) {
        out[num] = 2;
        return;
    }
    sucesor.h = heuristic(infonodes, sucesor.type, idEnd);
    sucesor.g = actual.g + search_cost_node_2_node(conexiones, nedges,
actual.type, sucesor.type);
    sucesor.f = sucesor.g + sucesor.h;
    bool flagSkip = false;
    j = 0;
    while (j < nabiertos) {
        if (abiertos[j].type == sucesor.type && abiertos[j].f <=
sucesor.f) {
            out[num] = 0;
            return;
        }
        j++;
    }
    j = 0;
    while (j < ncerrados) {
        if (cerrados[j].type == sucesor.type && cerrados[j].f <=
sucesor.f) {
            out[num] = 0;
            return;
        }
        j++;
    }
    sucesores[num] = sucesor;
    out[num] = 1;
}
```

Cada hilo escoge un sucesor relacionado con su identificador. Calcula la estimación de coste *h* y el coste real *g*. Después busca entre los nodos que se hayan en la lista de nodos abiertos y cerrados si ya existe alguno con una estimación *f* mejor (menor) que la suya. Si es así, el nodo ya no nos interesa puesto que, o bien se encuentra para explorar uno mejor que él o ya hemos explorado uno mejor, y lo marcamos como tal. Si no es así, el nodo es candidato a ser explorado y lo marcamos.

Kernel B1

```
// Implementación B1: algoritmo A* íntegro en GPU
__kernel void searchastar(__global infonode *infonodes,
                          __global edge *conexiones,
                          __global node *abiertos,
                          __global node *cerrados,
                          __global node *sucesores,
                          __global ulong *nlongs,
                          __global node *out,
                          __global int *out_state,
                          const ulong nnodos,
                          const ulong nedges,
                          const ulong idStart,
                          const ulong idEnd)
```

En esta implementación, en un principio, no sería necesario pasarle como parámetros los búferes de *abiertos* y *sucesores* pues sus datos no nos interesan fuera de la ejecución del algoritmo dentro del *kernel* (*cerrados* si nos interesa ya que a partir de los nodos insertados en él podremos reconstruir el camino óptimo). De todas maneras, se los pasamos como parámetros porque esta es la única manera de tener *arrays* dinámicos. Si no lo hiciésemos así, sólo podríamos crear listas de elementos de un único tamaño que no podría depender de *nnodos* (cantidad de nodos del grafo).

En *nlongs* hemos agrupado las variables *ncerrados*, *nabiertos* e *indexnodes*.

Mediante *out* devolveremos el nodo final, si se ha encontrado. *Out_state* indica el estado del algoritmo. 1 y 0 informan si el algoritmo ha encontrado camino o no, respectivamente. 2 es para indicar que el *kernel* debe ser llamado otra vez para continuar la ejecución de algoritmo. Esto ha sido necesario debido a que, con grafos muy densos o grandes, la búsqueda tarda en ejecutarse más de lo habitual haciendo que el *driver* de la GPU pare de funcionar durante la ejecución porque el Sistema Operativo no ha recibido ninguna señal suya. Esto causa que haya que reiniciar el sistema para que el *driver* se reinicie.

```
//Thread principal
    if(num == 0){
        out_state[0] = 0;
        if(nabiertos == 0){//Si es la 1ª iter, iniciamos nodo inicial
            inicial.type = idStart;
            indexnodes++;
            inicial.id = indexnodes;
            inicial.g = 0;
            inicial.parent = 0;
            abiertos[nabiertos++] = inicial;
        }
        while (nabiertos > 0 && !found) {
            reps++;
            if(max <= reps){
                break;
            }
            actual = abiertos[nabiertos-1];
            nabiertos--;
            /*Generamos sucesores*/
            nsucesores = genera_sucesores(sucesores, conexiones,
actual, nedges, indexnodes);
            indexnodes += nsucesores;
            if (nsucesores == 0) {
                continue;
            }
            /* Expandimos cada sucesor*/
            i = 0;
            while (i < nsucesores) {/*Código igual a A3*/}
            nsucesores = 0;
            cerrados[ncerrados++] = actual;
            bubblesort(abiertos, nabiertos);
        }
        /*Resultados*/
        if(max <= reps){
                nlongs[0] = nabiertos;
                nlongs[1] = ncerrados;
                nlongs[2] = indexnodes;
                out_state[0] = 2;
        }
        else{
            if(found){
                out[0] = sucesor;
                out_state[0] = 1;
            }
            else{
                out_state[0] = 0;
            }
        }
    }
```

Con la primera condición hacemos que un único hilo ejecute el algoritmo y, por tanto, el tiempo de ejecución total dependerá de dicho hilo. No hemos puesto el código de la expansión de nodos porque es el mismo de la implementación A3. A excepción de la comprobación sobre si ha llegado al máximo de iteraciones (*if(max <= reps)*) y de la parte final donde damos valores a las variables de respuesta, el resto del código es igual a la versión íntegra en CPU.

Kernel B2

Esta implementación ha sido la más conflictiva tanto en el proceso de su diseño como en su codificación. Una vez la codificación fallaba, había que volver a cambiar su estructura ya que aprendíamos algo nuevo relacionado con el comportamiento de los hilos de GPU.

En un principio, era interesante que se asemejase esta implementación lo máximo posible a A3. Un hilo (id = 0) sería el encargado de hacer de *host* mientras los demás esperarían sus indicaciones para expandir nodos. Lanzaríamos una cantidad de hilos igual a *nnodos*. Ya que un nodo tendría como máximo *nnodos-1* conexiones, serían suficientes hilos. Tras varios intentos, nos dimos cuenta de que era imposible. O bien la lectura y escritura de variables entre el hilo principal y uno de los hijos producía una condición de carrera o bien no podíamos utilizar las funciones provistas por OpenCL para sincronización porque sólo funcionan para aquellos hilos agrupados en un *WorkGroup* y no para todos los hilos.

Después de ver que era necesario usar un número de hilos limitado, tratamos de optimizar su uso. Cada vez que un hilo terminase con uno de los nodos sucesores, buscaría nodos que no hubiesen sido asignados y los analizarían. Mientras tanto, el hilo principal añadiría los nodos sucesores a los búferes correspondientes a medida que los hilos terminasen con ellos. Sin embargo, esta implementación también falló porque requería sincronización con una función de OpenCL llamada *barrier()* que detiene la ejecución de los hilos que pasen por ella hasta que todos los hilos del *WorkGroup* hayan llegado a esta instrucción. El problema que surgió es que era necesario que los hilos de los nodos sucesores fuesen sincronizados en un lugar distinto del código que el hilo principal. Por desgracia, el uso de esta función prohíbe eso y es necesario que todos los hilos ejecuten cada *barrier()* en el mismo lugar exacto de su código.

Por último, llegamos a la implementación actual.

```
// Implementación B2: algoritmo A* íntegro en GPU con
// paralelización de la expansión de nodos
__kernel void searchastar(__global infonode *infonodes,
                          __global edge *conexiones,
                          __global node *abiertos,
                          __global node *cerrados,
                          __global node *sucesores,
                          __global int *info_threads,
                          __global node *actual,
                          __global ulong *nlongs,
                          __global int *out_state,
                          __global node *out_result,
                          const ulong nnodos,
                          const ulong nedges,
                          const ulong idStart,
                          const ulong idEnd)
```

Tenemos un nuevo búfer llamado *info_threads* con el que poder pasar información del nodo sucesor durante la expansión de nodos al hilo principal.

En esta implementación, el código ha sido dividido por varias llamada a *barrier()* para tener controlado el flujo del programa y los hilos.

```
    //Thread principal
    if(num == 0){
        found = false;

        if(nlongs[0] == 0){
            inicial.type = idStart;
            nlongs[3]++;
            inicial.id = nlongs[3];
            inicial.g = 0;
            inicial.parent = 0;

            abiertos[nlongs[0]++] = inicial;
        }
    }

    barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE); //1 barrier
```

El hilo principal es el único que inicia variables del algoritmo.

```
barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE); //1 barrier

    while(globalReps < max){
        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE); //2
barrier
        if(nlongs[0] == 0 || found){
            break;
        }
        if(num == 0){
            actual[0] = abiertos[nlongs[0]-1];
            nlongs[0]--;
            nlongs[2] = genera_sucesores(sucesores, conexiones,
actual[0], nedges, nlongs[3]);
            nlongs[3] += nlongs[2];
            if (nlongs[2] != 0) {
                for(i = 0; i < nlongs[2]; i++){
                    info_threads[i] = 3;
                }
            }

        }
        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE); // 3
barrier
```

Hemos intercambiado las condiciones de paradas porque de esta manera era más simple controlar la condición de parada principal (nodo encontrado o lista de nodos abierta vacía). Las variables *nabiertos*, *ncerrados*, *nsucesores* e *indexnodes* son usadas directamente con el búfer *nlongs* (están posicionadas en ese mismo orden) para que los hilos hijo puedan acceder también a ellas.

El hilo principal genera sucesores e indica a los hilos hijo que pueden comenzar a recorrerlos.

```
barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE); // 3 barrier
        globalReps++;
        if(nlongs[2] == 0){
            continue;
        }
        if(num != 0){ //Threads para sucesores
            i = j = k = 0;
            numnodes = 0;
            nsucesores = nlongs[2];
            numnodes = (int) (nsucesores/(localSize-1));
            if((nsucesores % (localSize-1)) >= num){
                numnodes++;
            }
            for(k=0; k<numnodes; k++){//Código igual a A3}
        }
        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE); // 4
barrier
```

Ahora que puede haber más sucesores que hilos, estos deberán encargarse a veces de más de un nodo. Todos harán una misma cantidad.

```
        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE); // 4
barrier

        if(num == 0){

            i = 0;
            while (i < nlongs[2]) {
                option = info_threads[i];
                if(option == 2){
                    found = true;
                    sucesor = sucesores[i];
                    nlongs[0]=0;
                }
                else if (option == 1){
                    abiertos[nlongs[0]] = sucesores[i];
                    nlongs[0]++;
                }
               i++;
            }
            cerrados[nlongs[1]] = actual[0];
            nlongs[1]++;
            bubblesort(abiertos, nlongs[0]);
        }
    }//while principal
```

El hilo principal recibe la información de los hilos hijo sobre qué nodos son candidatos y qué nodos no, o si se ha encontrado el nodo final.

La parte final del algoritmo de retorno de resultados es muy similar a la ya vista en la implementación B1.

Kernel C

Esta última implementación se basa en su gran mayoría en B1, pero serán ejecutadas varias instancias del algoritmo. Habría sido interesante poder llamar desde el *host* a varios *kernel* de B1 que sean ejecutadas mientras haya unidades computacionales libres. Extrañamente, no es tan simple y hemos tenido que encontrar otra manera de hacerlo. Esta manera ha sido realizar todas las instancias en un mismo *kernel*. Básicamente, multiplicamos la cantidad de memoria pasada por parámetros a B1 por *numInstances* y dentro de él hilos diferentes se encargarán de ejecutar una instancia y acceder a aquellas zonas de memoria que pertenecen a la instancia asignada.

```
// Implementación C: varias instancias del algoritmo
// A* íntegro en GPU (B1)
__kernel void searchastar(__global infonode *infonodes,
                          __global edge *conexiones,
                          __global node *abiertos,
                          __global node *cerrados,
                          __global node *sucesores,
                          __global ulong *nlongs,
                          __global node *out,
                          __global int *out_state,
                          const ulong nnodos,
                          const ulong nedges,
                          const ulong idStart,
                          const ulong idEnd,
                          const int numInstances)
```

Tenemos un parámetro adicional indicando el número de instancias a ejecutar *numInstances*. Desafortunadamente, si el valor de este parámetro es el único que cambia, no hará que se ejecuten más o menos instancias. Es necesario que todos los búferes cambien de tamaño acordemente.

```
    //Threads para instancias
    if(num < numInstances){
        offset = (num*nnodos);
        offsetAbiertos = (num*nnodos*10);
        nabiertos = nlongs[(num*3)];
        ncerrados = nlongs[(num*3)+1];
        indexnodes = nlongs[(num*3)+2];
```

```
abiertos[offsetAbiertos+nabiertos] = inicial;
```

```
if (abiertos[offsetAbiertos+j].type == sucesor.type &&
abiertos[offsetAbiertos+j].f <= sucesor.f) {
                        flagSkip = true;
                        break;
                    }
```

```
sucesor = sucesores[offset+i];
```

En estos ejemplos, podemos ver cómo, dependiendo de la instancia que sea, se acceden y actualizan diferentes partes de los búferes.

offsetAbiertos tiene un valor diferente debido a que el búfer de los nodos abiertos hay casos puntuales donde, si el grafo es muy denso, sobresale del tamaño estándar de *nnodos*. Por ello ha sido necesario asignarle un valor más alto de lo normal.

```
nsucesores = genera_sucesores_range(sucesores, conexiones, actual,
nedges, indexnodes, offset);
```

```
bubblesort_range(abiertos, offsetAbiertos, offsetAbiertos+nabiertos);
```

También ha sido necesario crear unas funciones modificadas de *genera_sucesores* y *bubblesort* para que trabajen sobre determinados rangos.

El resto del algoritmo es similar a B1.